**A Step towards Static Script Malware Abstraction: Rewriting Obfuscated Script with Maude**

Gregory BLANC(a), Nonmember and Youki KADOBAYASHI(b), Member

**SUMMARY** Modern web applications incorporate many programmatic frameworks and APIs that are often pushed to the client-side with most of the application logic while contents are the result of mashing up several resources from different origins. Such applications are threatened by attackers that often attempt to inject directly, or by leveraging a stepstone website, script codes that perform malicious operations. Web scripting based malware proliferation is being more and more industrialized with the drawbacks and advantages that characterize such approach: on one hand, we are witnessing a lot of samples that exhibit the same characteristics which make these easy to detect, while on the other hand, professional developers are continuously developing new attack techniques. While obfuscation is still a debated issue within the community, it becomes clear that, with new schemes being designed, this issue cannot be ignored anymore. Because many proposed countermeasures confess that they perform better on unobfuscated contents, we propose a 2-stage technique that first relieve the burden of obfuscation by emulating the deobfuscation stage before performing a static abstraction of the analyzed sample’s functionalities in order to reveal its intent. We support our proposal with evidence from applying our technique to real-life examples and provide discussion on performance in terms of time, as well as possible other applications of proposed techniques in the areas of web crawling and script classification. Additionally, we claim that such approach can be generalized to other scripting languages similar to JavaScript.

key words: web-based malware, JavaScript, obfuscation, emulation, functional unit abstraction

1. Introduction

In an era where attack code is more and more industrialized, the number of attack websites keep increasing while new techniques are introduced to defeat state-of-the-art countermeasures. Though it seems we’ve reached a limit in the sophistication of attacks written in JavaScript (JS) as we keep on witnessing the same techniques applied here and then, new techniques are continuously being developed as reported by [1]. Not only the vectors do vary, ranging from malicious plugins to infected stylesheets to malware-embedded PDF files, but the languages employed are not solely restricted to JS and we are witnessing more and more Flash or Java-based attacks. This diversification advocates for a solution that does not only focused on accommodating JS but also extends its scope to other widely-used web scripting languages.

On the other hand, there have been recent accounts on obfuscation schemes that can evade currently deployed defenses, and that surpass common obfuscation techniques such as eval unfolding. In particular, Yosuke Hasegawa, the creator of jjencode [2], a popular JS encoding technique that uses only non-alphanumeric characters, has discovered some recent attacks were using his tool [3]. Such techniques are not considered by actual analysis tools and there is thereby a need for a more flexible tool capable to reverse such encoding schemes.

Our proposal focuses on client-side script-based malware and attempts to offer an alternative to dynamic analysis tools or offline malware analyzers that fail to protect the user. Although, some recent works have proposed real-time client-side analysis, we advocate a 2-stage technique that has the potential to significantly reduce false positives by providing comprehensive deobfuscation. Our proposed technique first cancel obfuscation using membership equational logic to emulate the deciphering routine. Then, it statically and abstractly models the intentions of the recovered script based on the functionalities the code expresses, enhancing the accuracy of the analysis.

This paper is outlined as follows: in Sect. 2, we will cover the targeted problem as well as common drawbacks seen in past proposals. Then, Sect. 3 will feature details about our proposed solution, supported by some concrete examples in Sect. 4. In Sect. 5, we will slightly discuss performance concerns of our proposals as well as possible other applications. Finally, Sect. 6 presents some noteworthy related works before we conclude in Sect. 7.

2. Problem Statement

Web 2.0 applications offer a dynamic and interactive user-experience thanks to numerous frameworks and APIs. These applications often mash code from several origins and accept content contributions from users. Applications such as social networks, collaboration platforms or mashups connect many individuals and often allow complex operations to be performed.

Developers have now less difficulties developing complex applications thanks to the above-mentioned frameworks and APIs. However, this simplicity partly obscures the application logic, on which developers have no more control. Much of the application logic is also pushed to the client-side, leading to processing overhead for the browser and possibly undesired application logic disclosure. Such disclosure may reveal business logic flaws that can be
abused by attackers to induce code injection, directly or indirectly affecting other end users. By abusing service-level agreements (SLAs) between applications belonging to different owners, an attacker can mount a pivot attack [4]: a vulnerable third-party application is exploited to inject code at the time it gets mashed into the target application.

To minimize the risk of being exposed, attackers favor browser-embedded technologies. Browsers natively integrate script engines and since developers make a great use of AJAX frameworks, it is not surprising to see many attack codes written in JS, a de-facto standard for AJAX applications. With users almost obliged to enable JS to enjoy a good user-experience, we have witnessed an increasing number of so-called JS malware, which is not often considered by state-of-the-art JS analysis approaches.

2.1 On JavaScript Malware

JavaScript malware (or JS malware) is a class of attacks that leverage the web browser’s capabilities to execute malicious script within the victim’s local execution context [5]. These are called malware since they tend to mimic the behavior of existing malware. They differ in that they need a JS-enabled environment to thrive and therefore propagate through the victim’s browser. JS malware can perform a large scope of intents ranging from drive-by attacks to XSS worms to XSS botnets. Such attacks are often characterized by the fact that:

- despite the possible initial phishing phase, the attack script itself keeps things low most of the time by leveraging the victim’s local execution context and objects such as the XMLHttpRequest object, essential to perform asynchronous HTTP communication;
- these scripts are not straightforward in the way they perform exploitation and the intents they carry are sometimes distributed to several files that may or may not be readily observable. Malicious code is often downloaded during a later stage through redirection or on-demand script loading, and possibly leveraging the mechanism of JavaScript clobbering (or known as Prototype hijacking for its malicious counterpart);
- the attack code may not be readily readable to the JS engine because it employs an obfuscation scheme that requires a deobfuscation stage before the script can be executed. However, character-encoding-based obfuscation does not need any deciphering routine and can be directly interpreted by the JS engine though being not human-readable.

Against such threats, recent proposals have concentrated on either sandboxing JavaScript [6], [7], instrumenting the JS engine or the browser [8]–[10], or learning features resulting from various analysis techniques [11]–[13]. We especially want to point out some drawbacks common to state-of-the-art proposals in the field of Web 2.0 security and thus advocate for a more appropriate solution.

2.2 Drawbacks of Dynamic Approaches

In program analysis, dynamic and static approaches have often been opposed and it is left at the option of the analyst to apply which fits best. It is commonly agreed that while dynamic approaches offer faster processing, these lack in code coverage which can be solved through several executions involving different inputs each time, degrading the time performance advantage. Another drawback is not linked to the supposed speed advantage of the dynamic program analysis approach: while it is obvious that obfuscated strings will eventually be deobfuscated in the process, it may be dangerous to assume that obfuscation and execution are not interleaved. But many proposals have also held that obfuscation was an indicator of malice while it has been demonstrated to be false [14]. On the contrary, other proposals have advocated the use of instrumentation to halt execution on deobfuscated strings, however, such instrumentation may have side-effects on the deobfuscation stage.

2.3 Drawbacks of Offline Approaches

One major characteristic of offline analysis services is obviously that it is directed to analysts. Users are not expected to test every web page they visit with offline analysis website before browsing the suspected page. In addition, such analysis may fail as it has been acknowledged that some anti-analysis techniques can defeat offline analysis. Redirect techniques used may also hinder analysis.

3. Proposal

The proposed system aims to fulfill some requirements necessary to accommodate JS malware and web-based scripting attacks in general:

- malicious JS code is always distributed to several sources and made available through several layers of redirection and obfuscation. The countermeasure needs to take into account such behavior by gathering files likely to contain malicious JS code;
- malicious JS code makes a heavy use of obfuscation, but obfuscation itself is not malicious. The countermeasure needs take steps to cancel obfuscation in order to safely decide on the nature of the analyzed script;
- malicious JS script may not directly exhibit malicious behavior or may not enable it though such malicious intents may be present in the code. The countermeasure needs to maximize the code coverage in order to detect such intents that dynamic analysis may miss;
- malicious JS is often short-lived or makes use of anti-analysis tricks to avoid being analyzed. The countermeasure needs to convey the browser environment without necessarily imposing risk to the end-user’s browser. Additionally, analysis should be done in realtime to better mimic the browser’s behavior.
3.1 Overview

Based on the above-mentioned requirements, we design our proposed system as a proxy-based solution (see Fig. 1) to intercept HTTP traffic and analyzed script contents in order to detect any malicious intent towards the user's browser. It enforces pre-fetching to overcome limitations imposed by malicious scripts that scatter their own code across several links, concealing their real intent. A parse tree is built from the aggregated contents and passed to a deobfuscator. After detecting which obfuscation scheme is employed, the script is reversed accordingly in order to provide an interpretable script upon which decision can be made. A proxy implementation avoids imposing security issues on the user's browser, unless it is bypassed, and allows alleviating computational overhead on the browser, already busy processing rich-internet application (RIA) contents.

3.2 Modules

The proposed system performs 3 main operations on each analyzed web page, which are the following:

- **pre-fetching**: as stated in Sect. 2.1, script-based malware are often concealing themselves through several layers of redirection and obfuscation. Therefore, the first thing we need to do is to make available scripting contents likely to contain malicious code to the analyzer. Malicious code can be scattered across DOM-embedded code, links, frames or even images or stylesheets. The pre-fetching module is responsible for detecting the possible locations of script contents and downloading these scripts all at once in order to provide an aggregated script to the next module;

- **deobfuscation**: at this stage, the aggregated script is likely to include obfuscated strings and possibly deciphering routines as well. Besides, the script itself may contain anti-analysis traps. However, to overcome these limitations, the script will not be executed as is, but rather processed by a high-level language in order to emulate the deciphering routine. This stage and the previous one can be repeated as many times as needed;

- **decision**: it is assumed that we can perform a more precise analysis of a deobfuscated script. This stage aims to abstract a script as a model that expresses its true intent. The script is decomposed into abstractions that each expresses a particular functionality. The combination or sequence of these functionalities might inform on possible malicious intents.

3.3 Recursive Pre-fetching of Suspicious Linked Contents

A malicious script is not expected to be monolithic and is often scattered across script files, DOM-embedded contents or iframes. In particular, linked contents injected via script or iframe tags, may originate from different domains. Therefore, the script found in the original web page may not be malicious itself, or hard to decide on. It is necessary to fetch these additional or linked contents in order to have an accurate view of the scripting contents involved in a web page. Another example is on-demand loading, where additional contents are only downloaded later during execution. Such behavior is implemented through the XmlHTTPRequest (XHR) object which allows asynchronous HTTP communication. Pre-fetching here will look for XHR and download script contents in advance to evaluate its impact on the current script contents. Prototype hijacking [15] has the potential to override a benign function with a malicious one.

Since malicious scripts may be obfuscated through several layers of obfuscations and redirections, the two first processing steps are recursive. Once a deobfuscation step has been taken, the outcome may be still obfuscated but the deciphering routine may not be directly present in the output but rather as a link, a DOM-embedded string or an iframe. Therefore, it is necessary to run the pre-fetching step again to gather scripting contents used in the deobfuscation step. These steps are repeated until the outcome is a plain interpretable script with no linked or DOM embedded contents.

3.4 Emulation-Based Deobfuscation

Obfuscation is any transformation that render a piece of code unreadable, hence hindering analysis by a human or an automated analyst. Obfuscating transformations span a
wide set of techniques ranging from simple string splitting to encryption.

Obfuscated scripts, in particular, malicious obfuscated scripts carry out 4 common stages as stated in [16]: 1) redirection and cloaking, 2) deobfuscation, 3) environment preparation and 4) exploitation. This highlights the fact that some obfuscated scripts, e.g., encrypted ones, will eventually deobfuscate themselves before further execution. Such argument is always pointed out by dynamic analysis advocates to support the fact that obfuscation is not an issue, if not completely discarded by researchers that confuse obfuscation with malware. But the boundary between deobfuscation and the following stages is not always clear and some recently witnessed obfuscation schemes do interleave obfuscation with environment preparation and/or exploitation.

Since static methods cannot overcome the obstacle of obfuscation, emulation allows to reproduce the outcome of the deobfuscation stage without incurring side-effects inherent to a real execution. Here, the emulation is not aimed at the whole script and should only be carried out on the obfuscated slice. The obfuscated slice is comprised of all the instructions involved in deobfuscating the script: it can be further decomposed into the obfuscated strings and the deciphering routine (or decoder), when it exists. The first step of deobfuscation is therefore to extract the obfuscated slice from the aggregated script parse tree, obtained after pre-fetching. But only the decoder is actually emulated, obfuscated strings being used as input.

In this research, we focus on JavaScript, which has been widely used in attack scenarios. JavaScript is both an object-oriented and functional language but the obfuscation schemes we are considering seldom make use of functional properties of JavaScript (variable promotion, variable substitution). Taken that the deobfuscation stage cancels a prior obfuscation to provide an executable script to the next stage, the deobfuscation process is bound to terminate. We assume that the deobfuscation stage ultimately converges to a unique normal form: the unobfuscated original script.

Concerned with issues of performance, soundness and completeness, we considered several approaches to automate the deduction of obfuscated strings by the deciphering routine. Emulating JS instructions through another scripting language obviously suffers from lack of completeness as well as poor performance inherent to scripting languages in general. To satisfy properties of soundness and completeness, we considered formal approaches such as theorem proving but state-of-the-art theorem provers can not completely emulate JS obfuscating transformations. Eventually, we turned to rewriting systems. Maude [17] is such rewriting framework which underlying logic is membership equational logic. Meseguer [18] observed that equational logic is very well suited to give executable axiomatizations of imperative sequential languages. It was suggested to us that functional modules in Maude [17] could fit our requirements for sound and complete deduction of the outcome of JS instructions. As a matter of fact, computation in functional modules is accomplished using the equations as rewrite rules.

Functional modules satisfy the membership equational logic as well as the additional requirement of being confluent and terminating. Functional modules are thereby used to emulate deobfuscation: variables and objects are mapped to Maude’s sorts, instructions are emulated through equations. Computation is realized by using these equations as rewrite rules applied to the obfuscated strings, until a canonical form is found, i.e., the deobfuscated script. Figure 2 depicts the extraction and conversion steps to deobfuscate a script: at first, a preliminary analysis should allow the system to disambiguate obfuscated contents from the rest of the code, and then further isolate the deciphering routine and the obfuscated strings. The deciphering routine is then converted to a functional module’s equations which are then used to rewrite the obfuscated string through reduction. Algorithm 1 gives a more detailed description of the processing that takes place just after pre-fetching. Once the obfuscation scheme employed has been detected (out of the scope here), the obfuscated path is extracted and the deciphering routine is converted into a Maude functional module. Upon deobfuscation, an additional step verifies whether deobfuscation is still needed.

An advantage of Maude is that it employs term-indexing techniques to achieve high speeds of rewriting [19]. However, it does not support every JS native construct, such as loops. Yet, we can take advantage of

---

**Algorithm 1** Automated deduction of script instructions

1: obfstring, decroutine = extract(script)
2: if script contains loops then
3: script = loopToRecursion(loops)
4: end if
5: fmod = convert(decroutine)
6: output = Maude.reduce(fmod, obfstring)
7: if output contains links then
8: output += prefetch(links)
9: end if
10: if output is obfuscated then
11: script = output
12: repeat from line 1
13: end if

---

![Fig. 2 Overall processing of analyzed scripts.](image-url)
conditional equations to emulate recursions. Doing so requires additional processing to transform JS loops to recursive functions.

3.5 Static Functional Unit Decomposition

Static methods are difficult to apply to obfuscated strings. Our method here applies to a deobfuscated string obtained after emulation. The intuition is that we are able to tell what the script intends to do without executing it. By looking to what the script offers to do, i.e., its functionalities, it is possible to understand the actions a script might perform upon execution. To that end, script contents undergo static flow analysis on function calls or operators that are traced back to the root of an instruction block. This analysis allows building functional units that decompose the script into blocks that express a single functionality. The combination or sequence of functionalities indicate what are the intentions of the script. A functional unit is a set of instructions that express a single functionality. The term was first coined by Lu and Kan [20] and originally refers to a JavaScript instance, combined with all of (potentially) called subprocedures. Algorithm 2 describes the steps taken to cluster instructions to functionalities and link these through their interaction. A block-based forward flow tracing approach is used to list native function calls in every block of the program as well as instructions related to these functions. Functions manipulating common variables are then seen as interacting, providing a logical link between two clusters.

While Lu and Kan favored a top-down approach, we advocate a bottom-up linking approach in order to focus on the functional characteristic of the clustered instructions. Besides, a JS function is not expected to perform a single functionality, especially malicious ones. Our method applies on the parse tree (or its abstract syntax tree (AST) representation) of a deobfuscated script and directly targets explicit functions. Explicit functions are functions of which functionality is obvious and that have been classified by us.

By tracing the flow of these functions, we can identify clusters of instructions that express a single functionality (Fig. 3) and eventually trace the flow of variables through these different clusters. Such abstract model can be then compared with a database of models. At the time of writing, we are not able to predict the average size of such models. Examples, as the ones we will present in Sect. 4, usually feature 3 or 4 functional units.

4. Example

In this section, we present an example of the application of our proposal to a simple malicious JavaScript sample. Although the whole procedure is not fully automated, this example will demonstrate the feasibility and the accuracy of our proposal.

This example (see Fig. 4) is a simple eval unfolding featuring a single loop that decipher an obfuscated string via XOR operations. The script is included into an HTML file that displays a 404 error page to an unsuspecting user.

The proposal first extracts the script contents, i.e., instructions comprised between the \texttt{<script>} tags, and parses the contents. The parse tree is analyzed to detect the obfuscation scheme. For the moment, we are considering the possibility that functional unit abstraction could be used to model obfuscation schemes and therefore be applied to detect the type of obfuscation and extract the obfuscated path. Here, the obfuscation scheme uses a loop to process the obfuscated string. This loop is converted to a recursive function whose body is the aforementioned string processing script. The Maude system readily provides a predefined functional module that defines the string data type as well as operators to manipulate string objects: the\texttt{fromCharCode()} function is mapped to Maude’s \texttt{char} operator, which converts an ASCII code to the corresponding character; the \texttt{charCodeAt()} function is emulated by the combination of two basic operators, \texttt{ascii}, the inverse of \texttt{char}, and \texttt{substr}, the substring operator. The result of the conversion to a Maude functional module is displayed in Fig. 5. The workflow of the recursion is realized through conditional equations. The obfuscated string \texttt{str} as well as the empty string \texttt{str2} are inputs to the Maude system and are going to be rewritten by the functional module we generate (Fig. 5). The output is then parsed and decomposed into functional units. The \texttt{open()} call issued by the variable \texttt{asq}, which is an instance of the \texttt{msxml2.XMLHTTP} object allows clustering instructions dedicated to the download of some data, while the \texttt{SaveToFile()} call from the

\begin{algorithm}[h]
\caption{Block-based forward flow tracing}
\begin{algorithmic}
\Function{functions} {[]} 
\Function{params} {[]} 
\Function{transitions} {[]} 
\For {$blk \in \text{blocks}$} 
\For {$instr \in \text{instructions}$} 
\If {$instr \geq \text{call} \in \text{calls}$} 
\Function{functions}[call].add($instr$, $blk$) 
\EndIf 
\EndFor 
\EndFor 
\For {$f \in \text{functions}$} 
\For {$instr, blk \in \text{f}$} 
\If {($instr, blk) \in \text{transitions}$} 
\Function{transitions}.add($instr$, $blk$) 
\EndIf 
\EndFor 
\EndFor 
\EndFor 
\EndFunction 
\EndFunction 
\EndFunction 
\Function{transitions}.sort()
\end{algorithmic}
\end{algorithm}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{plot.png}
\caption{The proposed JavaScript functional unit.}
\end{figure}
ADODB stream instance, `asst` isolates storage-related instructions. Finally, the `shellexecute()` function is linked to a `ShellApplication` instance, `ass` that expresses an execution functionality. The clustered instructions have been colored differently in a modified output (see Fig. 6) of the Maude framework. This output can be further abstracted to a functional model as shown in Fig. 7. This abstract view offers a more straightforward model of what activities the malicious script is actually carrying out: after downloading contents from a remote URL, the contents of the response, `asq.responseBody`, are passed to a storage functionality that saves these into a file called `imya` and this file is then inputted into an execution functionality.

5. Discussion

5.1 Performance

We have previously opposed static and dynamic analysis approaches in Sect. 2.2, especially on the grounds of time performance. In our approach, we wish to maximize code coverage by adopting a rather static approach for script analysis. This needs prior deobfuscation of analyzed samples. This preprocessing stage surely incurs some delay and emulating the deobfuscation stage using the Maude framework may minimize the time overhead. As a matter of fact, our approach features some time-saving points to reduce as much as possible the delay: a) the prefetching stage anticipates the fact that several snippets of code will be gradually downloaded to the client-side; b) the conversion stage between script contents and Maude functional modules is expected to be fast as we are mapping script objects to Maude predefined data types and operators; c) Maude rewriting logic performs well as indicated by [19]. Additionally, we have conducted some preliminary tests using single-obfuscated samples. We manually converted their deobfuscation routine and ran the Maude 2.5 engine on an Intel Core 2 Duo platform (2.53 Ghz) with 4 GB of memory. Rewriting overhead was around 100 milliseconds, which is low compared to time overhead loading RIA applications. However, this does not take into account several intermediate processing such as conversion time between JS instructions and Maude functional modules, and only rewrites a single obfuscation.

We are aware that our claims still need to be confirmed by actual experimentation once we will have completed the development of a full-fledged system. Still, we hold expectations that the whole system time overhead be acceptable compared to how current client-side scripts perform.

5.2 Applications

The proposed system features modules that can find applications in some other areas. In particular, crawlers often struggle to collect data from attack websites since some are not able to handle script contents. By combining our deobfuscation module with a web crawler, we expect the crawler to deobfuscate code and fetch additional contents from collaborating remote attack hosts and thus coming with a more complete view of web attack strategies.

On the other hand, the functional unit decomposition can be generalized to models of benign scripts and therefore be employed to classify scripts as it has been done in [20].
Such system would be implemented using machine learning methods in order to automate classification. Overall, the contributions of this paper should not be understood as restricted to the JavaScript language, but are to be seen as flexible enough to be adapted to similar script languages such as VBScript or ActionScript. Indeed, by elaborating semantics for other languages than JavaScript, we can easily accommodate script samples written in other web scripting languages.

6. Related Works

6.1 Dynamic Analysis

First attempts to tackle web-based attacks have seen proposals such as [21] where the suspected web page is being executed in a proxy-based sandboxed virtual machine. Based on the execution results, the web page was allowed to be requested again or not by the end-user’s browser. The method is straightforward and should not let any doubt on the harmfulness of the page. However, the authors recognized that non-deterministic contents, where malicious scripts would be included occasionally, could defeat their method in the case the end-user were to download a malicious page after the system has been previously presented with a benign web page. Other dynamic analysis proposals, based on machine learning methods, have demonstrated potential to detect malice in web pages though they often confuse malice with obfuscation [11], [12]. Such misconception has, however, been dispelled since 2007 by Provos et al. [14] in their survey of web-based malware.

6.2 Offline Analysis

Web malware analysts are also familiar with malware analysis online services such as VirusTotal [22], WEPAWET [23] or JSunpack [24]. VirusTotal offers to submit a file or URL and then gives access to the reports of numerous antivirus vendors as well as some analysis tools. Although, the service seems to offer a rather exhaustive analysis of submitted samples, most anti-virus vendors are using static signature-based engines which is quite ineffective against obfuscated malware. Additionally, some security researchers have reported that most of these engines do report obfuscation as malice leading to a high rate of false positives [25].

WEPAWET and JSunpack are much more fit for the task of analyzing JavaScript or Flash contents and benefit from expertise of researchers of the field. However, as offline analyzers, they are vulnerable to anti-analysis tricks when submitted samples lack from a context. Overall, such offline analyzers are very useful to security analysts but are not fit to protect the end-user which is our goal here.

6.3 Hybrid and Online Approaches

Recently, some proposals have been made towards the integration of script analysis systems into the client-side environment or the browser in order to assure the end-user protection. Interestingly, such proposals leverage a hybrid analysis, i.e., it combines both static and dynamic analysis.

Cujo [26] is a web proxy that detects drive-by-download attacks. It leverages a hybrid analysis to produce word-based features. The feature extraction method is generic as it scans both static and dynamic analysis reports to detect characteristic patterns of q words where q is an integer. Static analysis is carried out by a JavaScript lexer and outputs JavaScript lexical tokens while dynamic analysis relies on Spidermonkey [27] to output variable manipulation and function calls as well as newly-added abstract operations (encoded as regular expressions and matched on-the-fly during dynamic analysis). Top contributing q-grams (q-uplets of words) are learned using Support Vector Machines. Their method tested against a dataset provided by the WEPAWET team demonstrates a detection rate of 94.4% and a false positive rate of 0.002%. Additionally, their system has a median operation run-time of 500 ms.

Zozzle [28] is based on previous research on Nozzle [29]. Nozzle was a defense against heap-spray attacks and Zozzle can be used along with Nozzle for better detection of drive-by-download attacks involving heap-spraying. The main goal is to integrate Zozzle into the JS parser of a browser to prevent heap-spraying attacks using a low-overhead mostly static detection method. The authors recommend their proposal as a first step to drastically reduce the expected end-user overhead for in-browser protection. It can also be used as an offline scanner. The authors characterize heap-spray scripts into 3 distinct parts: the shell-code, the spraying code and the vulnerability triggering code. Zozzle observes the different stages of deobfuscation by instrumenting the jscript.dll of the IE browser. By doing so, a script can be represented as an unfolding tree made of the different contexts resulting from eval unfolding. Zozzle is therefore able to collect informations on which context did generate which context and how context do rely upon other contexts. Zozzle must be run on unobfuscated JavaScript as stated by the authors, as they leverage the fact that JS needs to deobfuscate itself to execute. They used a Bayesian classifier to distinguish malicious from benign scripts. The accuracy of their method never goes below 97% and reaches a a false positive rate comprised between 4.5% and 0.01% at best using automatic feature selection, while the false negative rate can be up to 11% for some extracted features. Integrating Zozzle into the JS results in a great time performance with an an average parse time of 0.86 ms.

7. Conclusion

This paper presents our proposal to tackle modern script-based malware in an efficient way to accommodate current issues in obfuscation and malicious intent detection. Inspired by previous works done in the field of JavaScript classification, we advocate a more static approach in malicious JavaScript analysis that has the potential to highlight a piece of malware’s functionalities. This approach can be...
generalized to both benign and malicious samples and also across other scripting languages similar to JavaScript. This paper also supports evidence that deobfuscation may be an important stage in malware analysis though many past research works have either tried to avoid this issue or have wrongly associated obfuscation with malware. The authors are well aware that further development and evaluation is needed to produce evidence supporting their claim and their future work will be dedicated to this task. In particular, they wish to demonstrate that their proposal does not suffer a too long time overhead, resulting in a rather usable measure.

Acknowledgements

We would like to thank Pr. Hiroyuki Seki and the reviewers for providing very valuable comments. This work is supported by a grant from the NEC C&C Foundation.

References

[1] T. Greene, “Top 10 Web hacking techniques of 2010 revealed,” http://www.networkworld.com/news/2011/012411-top-web-hacking-techniques.html
[2] Y. Hasegawa, “Jencode - encode any javascipt program using only symbols,” http://inf-8.jp/public/jencode.html
[3] exploitdev, “Dollars javascript code - yet another Javascript obfuscation method for cc frauds,” http://extraexploit.blogspot.com/2010/10/dollars-javascript-code-yet-another.html
[4] J. Grossman, “Web 2.0 Pivot attacks,” http://jeremiahgrossman.blogspot.com/2010/02/web-20-pivot-attacks.html, 2010.
[5] M. Johns, “On JavaScript malware and related threats,” J. Comp. Vnr., vol.4, no.3, pp.161–178, Aug. 2008.
[6] M.S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, “Safe active content in sanitized javascript,” Tech. Rep., Google, Inc., 2008.
[7] S. Maffeis and A. Taly, “Language-Based isolation of untrusted javascript,” Proc. 22nd IEEE Comp. Sec. Found. Symp., 2009.
[8] C. Reis, J. Dunagan, H.J. Wang, O. Dubrovsky, and S. Esmeir, “Browsershield: Vulnerability-driven filtering of dynamic html,” Proc. 7th Symp. Op. Sys. Design and Impl., 2006.
[9] D. Yu, A. Chandler, N. Islam, and I. Serikov, “Javascript uninstrumentation for browser security,” Proc. 34th Annual ACM Symp. Princ. of Prog. Lang., 2007.
[10] C. Grier, S. Tang, and S. King, “Secure web browsing with the op web browser,” Proc. 2008 IEEE Symp. Sec. and Priv., 2008.
[11] P. Likarish, E. Jung, and I. Jo, “Obfuscated malicious javascript detection using classification techniques,” Proc. 4th International Conf. Malicious and Unwanted Soft., 2009.
[12] Y. Choi, T. Kim, S. Choi, and C. Lee, “Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis,” Proc. 1st International Mega-Conf. Fut. Gen. Inf. Tech., 2009.
[13] Y.T. Hou, Y. Chang, T. Chen, C.-S. Laih, and C.-M. Chen, “Malicious web content detection by machine learning,” Expert Syst. Appl., vol.37, no.1, pp.55–60, Jan. 2010.
[14] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, “The ghost in the browser analysis of web-based malware,” Proc. 1st Work. Hot Topics in Under. Botnets, 2007.
[15] S. Di Paola and G. Fedon, “Subverting AJAX,” Proc. 23rd Chaos Comm. Congress, 2006.
[16] M. Cova, C. Kruegel, and G. Vigna, “Detection and analysis of drive-by download attacks and malicious javascript code,” Proc. 19th International WWW Conf., 2010.
[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C.L. Talcott, “The maude system,” http://maude.cs.uiuc.edu/
[18] J. Meseguer, “Software specification and verification in rewriting logic,” 2003.
[19] N. Shankar, “Automated deduction for verification,” ACM Comput. Surv., vol.41, no.4, Oct. 2009.
[20] W. Lu and M.Y. Kan, “Supervised categorization of javascript using program analysis features,” Proc. 2nd Asian Inf. Retrieval Symp., Springer-Verlag, 2005.
[21] A. Moshchuk, T. Bragin, D. Deville, S.D. Gribble, and H.M. Levy, “SpyProxy: Execution-based detection of malicious web content,” Proc. 16th Annual USENIX Sec. Symp., 2007.
[22] Hispacec Sistemas, “Virus total - free online virus, malware and URL scanner,” http://www.virustotal.com/
[23] S. Ford, M. Cova, C. Kruegel, and G. Vigna, “WEPAWET,” http://wepawet.cs.ucs.edu/index.php
[24] B. Hartstein, “Jsunpack: An automatic javascript unpacker,” Shmoocon, 2009.
[25] M. Sutton, “Antivirus struggling with obfuscated javascript,” http://research.zscaler.com/2010/06/antivirus-struggling-with-obfuscated.html
[26] K. Rieck, T. Krueger, and A. Dewald, “Cujo: Efficient detection and prevention of drive-by-download attacks,” Proc. 26th Annual Comp. Sec. Appl. Conf., 2010.
[27] B. Eich, “SpiderMonkey (JavaScript-C) engine,” http://www.mozilla.org/js/spidermonkey/
[28] C. Cursinger, B. Livshits, B. Zorn, and C. Seifert, “Zozzle: Low-overhead mostly static javascript malware detection,” Tech. Rep., Microsoft Research, 2010.
[29] P. Ratanaworabhan, B. Livshits, and B. Zorn, “Nozzle: A defense against heap-spraying code injection attacks,” Tech. Rep., Microsoft Research, 2008.

Gregory Blanc received his Master degree in Network and Information Security from École Supérieure d’Informatique Électronique Automatique (ESIEA) in 2008. He is currently a Ph.D. student in the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. His research interest includes web application security, program analysis and machine learning.

Youki Kadobayashi received his Ph.D. degree in Computer Science from Osaka University, Japan, in 1997. He is currently an Associate Professor in the Graduate School of Information Science, Nara Institute of Science and Technology, Japan. Since 2009, he has also been working as an associate rapporteur of ITU-T Q.4/17 for cybersecurity standardization. His research interest includes cybersecurity, web application security, and distributed systems. He is a member of ACM and IPSJ.