Identifying Evasive Code in Malicious Websites by Analyzing Redirection Differences*

Yuta TAKATA†(a), Nonmember, Mitsuaki AKIYAMA†, Takeshi YAGI†, Takeo HARIU†, Kazuhiko OHKUBO†, and Shigeki GOTO††, Members

SUMMARY Security researchers/vendors detect malicious websites based on several website features extracted by honeyclient analysis. However, web-based attacks continue to be more sophisticated along with the development of countermeasure techniques. Attackers detect the honeyclient and evade analysis using sophisticated JavaScript code. The evasive code indirectly identifies vulnerable clients by abusing the differences among JavaScript implementations. Attackers deliver malware only to targeted clients on the basis of the evasion results while avoiding honeyclient analysis. Therefore, we are faced with a problem in that honeyclients cannot analyze malicious websites. Nevertheless, we can observe the evasion nature, i.e., the results in accessing malicious websites by using targeted clients are different from those by using honeyclients. In this paper, we propose a method of extracting evasive code by leveraging the above differences to investigate current evasion techniques. Our method analyzes HTTP transactions of the same website obtained using two types of clients, a real browser as a targeted client and a browser emulator as a honeyclient. As a result of evaluating our method with 8,467 JavaScript samples executed in 20,272 malicious websites, we discovered previously unknown evasion techniques that abuse the differences among JavaScript implementations. These findings will contribute to improving the analysis capabilities of conventional honeyclients.

key words: malicious website, redirection, javascript, evasive code

1. Introduction

Drive-by download attacks that infect clients with malware through the Web are continuously evolving. When a client accesses a landing website to be the source of an attack, it is redirected to malicious websites via multiple websites, called a “redirection chain.” In malicious websites, attack code that exploits the vulnerabilities of browsers and their plugins is executed, and the redirected client is forced to be infected with malware [2]. Security researchers have proposed methods of detecting malicious websites using high-interaction honeyclients [3]–[5]. A high-interaction honeyclient is a decoy system using a real browser that detects exploitations and malware downloads by monitoring unintended processes and filesystem accesses. However, attackers improve the exploitation success rate by identifying the client environment, i.e., OSes, browsers, and plugins, with browser fingerprinting and redirecting only targeted/vulnerable clients to malicious websites [6]. Such environment-dependent attacks make our analysis difficult since high-interaction honeyclients are not redirected to malicious websites when their fingerprints, e.g., the UserAgent strings and plugin-version values, do not match the attackers’ targets. In comparison, a low-interaction honeyclient that uses a browser emulator as a decoy system can emulate arbitrary client environments and collect detailed information of websites such as its document object model (DOM) tree, JavaScript execution traces, and redirections [7]. Using these features collected using low-interaction honeyclients, many researchers have proposed various detection methods of malicious websites by signature matching and machine learning [8]–[11]. Therefore, maximizing the exposure of malicious websites, i.e., triggering redirections and extracting features, is important for detection with low-interaction honeyclients. However, environment-dependent attacks have become more sophisticated along with the development of these analysis and detection methods. Attackers evade our analysis at the same time as identifying client environments by abusing the differences among JavaScript implementations such as browser-specific functions and quirks [12], [13]. Although simple environment-dependent attacks directly identify client environments with browser fingerprinting to exploit only vulnerable clients, attacks with evasion techniques indirectly identify them to avoid our analysis, especially automated analysis using low-interaction honeyclients. Many current malicious websites are deployed using exploit kits, and attackers can easily construct a malware-distribution network by only setting targeted clients, exploit code, and malware through these kits [14]. The above evasion technique is also known to be distributed to malicious websites through these kits, and other exploit kit families borrow evasive code from each other [15]. Therefore, we must immediately address this problem so that low-interaction honeyclients can analyze pervasive evasive code and extract features.

In this paper, we propose a method of extracting evasive code by analyzing HTTP transactions (HTTP requests and responses) obtained using two types of clients, a real browser and browser emulator. Our method leverages the evasion nature, i.e., the results in accessing malicious websites by using a real browser are different from those obtained using a browser emulator. More precisely, we collect two types of HTTP traffic of the same website using a
real browser and browser emulator that emulates the same client environment of the real browser. These browsers have the same UserAgent strings and plugin-version values but different JavaScript implementations. Our method extracts evasive code by leveraging accessed URL mismatches (redirection differences) in the HTTP traffic pair due to the above implementation differences. After reducing the number of extracted evasive code by code clustering, we manually identify the evasion techniques. We evaluated our method with 20,272 HTTP traffic pairs and 8,467 JavaScript samples of malicious websites observed during a four-year period. As a result, we extracted 281 evasive-code candidates from 8,467 pieces of JavaScript code and identified five evasion techniques that abuse the differences among JavaScript implementations from the candidates. In addition, five bugs in the browser emulator were discovered as a by-product. These findings will contribute to improving the analysis capabilities of conventional low-interaction honeyclients.

In summary, we make the following contributions.

- We propose a method of extracting evasive code through a differential analysis of HTTP transactions obtained using a real browser and browser emulator.
- As a result of classifying JavaScript code executed in malicious websites with our method, we discovered five previously unknown evasion techniques that abuse the differences among JavaScript implementations.
- We show that a part of evasive code can be used for effective signatures to detect malicious JavaScript code.

The rest of this paper is structured as follows. In Sect. 2, we provide examples to illustrate problems with evasive code. We introduce our proposed method in Sect. 3. We explain an experiment conducted to evaluate our method in Sect. 4 and present case studies on our findings in Sect. 5. We discuss the limitations of our method in Sect. 6 and review related work in Sect. 7. We conclude the paper in Sect. 8.

2. Motivating Example

Attackers abuse various web techniques to evade analysis and detection by security researchers/vendors. For example, JavaScript code pieces separately written in many script tags (scattered code) and JavaScript code dynamically generated by eval() and DOM manipulation functions (obfuscated code) are used in malicious websites to evade signature matching [16]. In addition, attackers abuse browser-fingerprinting techniques to increase the success rate of exploitation. Browser fingerprinting, which is a method of identifying a client environment, is generally used for user tracking and distributing web content according to the environment. Attackers leverage browser fingerprinting to redirect only vulnerable clients to subsequent malicious URLs on the basis of the client’s fingerprint in the middle of the redirection chain [6]. The code snippet in Fig. 1 redirects Internet Explorer (IE) 8 to a malicious URL with the domain name malicious.example and does not redirect the other browsers by comparing the UserAgent strings. Such a technique is also abused for circumventing the detection of security researchers/vendors by redirecting them to a benign URL or responding with empty content, called “cloaking” [17].

Along with the sophistication of these targeting techniques, security researchers have proposed various methods of JavaScript code analysis and machine-learning detection with low-interaction honeyclients such as browser emulators and content analyzers [8]–[11]. However, attackers have also begun to evade automated analysis in addition to identifying client environments [12]. For example, attackers use sophisticated JavaScript code that abuses the differences between browser implementations such as those in exception handling and JavaScript implementations [13]. Most malicious websites are deployed using exploit kits, and evasive code is also distributed to each malicious website through these kits [15]. An exploit kit called “Angler EK” is known to use evasive code, as shown in Fig. 2 and attempts to avoid the analysis of low-interaction honeyclients that are usually designed to provide a valid ActiveXObject at all times by intentionally throwing an error of ActiveXObject for the try clause [18]. If the purpose of attackers is to exploit only vulnerable clients, it is sufficient to use conventional browser fingerprinting for attacks. In other words, indirectly identifying clients by using evasive code means that the purpose is to evade analysis with low-interaction honeyclients. In fact, we confirmed that conventional methods using a browser emulator [8]–[11] cannot analyze the evasive code discovered in this paper.

Our objective is to extract evasive code, understand the current evasion techniques, and improve the analysis capabilities of low-interaction honeyclients. Our method contributes to solving the commonly known problem of conventional methods that attackers can easily detect and evade due to the low-interactivity. Note that we call JavaScript code that abuses the differences among JavaScript implementations “evasive code” and JavaScript code that checks the UserAgent strings and plugin-version values “browser-fingerprinting code”.

```javascript
var ua = navigator.userAgent;
var d = window.document;
if(ua.indexOf("MSIE 8") > -1) {
  var iframe = d.createElement("iframe");
  iframe.setAttribute("src", "http://malicious.example/");
  d.body.appendChild(iframe);
}
```

Fig. 1 Redirection code with browser fingerprinting

```javascript
try {
  new ActiveXObject("dunny");
} catch (e) {
  location.href = "http://malicious.example/";
}
```

Fig. 2 Redirection code with evasion technique
3. Proposed Method

Our method analyzes HTTP traffic pairs of the same website obtained using a real browser and browser emulator that emulates the same client environment of the real browser. Therefore, these browsers have the same UserAgent strings and plugin-version values but different JavaScript implementations. We can extract evasive code that abuses the differences among JavaScript implementations by leveraging the above differences. Moreover, we can improve the efficiency of analysis by code clustering to identify the evasion techniques by manually analyzing the extracted code.

3.1 Overview

Figure 3 shows an analysis pipeline of our method. The input data of the pipeline is two kinds of traffic data. The first one is real-browser traffic with malicious URLs, i.e., evasive code was executed, and another is browser-emulator traffic without malicious URLs, i.e., evasive code was not executed. First, our method identifies a candidate URL of a website that may contain evasive code by matching these HTTP transactions (Sect. 3.2). Next, pieces of JavaScript code in the candidate URL are extracted as evasive-code candidates and concatenated with JavaScript context for handling scattered/obfuscated code (Sect. 3.3). Finally, our method classifies the concatenated code using a clustering algorithm to reduce the number of evasive-code candidates for subsequent manual analysis (Sect. 3.4).

3.2 Redirection-Path Matching

The redirection-path matching component of our method identifies a candidate URL of a website that may contain evasive code. First, redirection graphs, in which the vertices represent URLs and the edges represent redirections, as shown in Fig. 4, are constructed using real-browser traffic (RealGraph) and browser-emulator traffic (EmuGraph), respectively. To construct a redirection graph, we use a method [19] for constructing link relationships using URLs in the Referer field or Location field of HTTP headers and URLs on HTTP bodies. Next, using these two graphs, our method identifies which redirection path in the EmuGraph corresponds to a redirection path that redirects to a malicious URL (a malicious path) in the RealGraph. More precisely, our method enumerates URLs from the root URL to the leaf URL on each redirection path in the EmuGraph. A redirection path with the largest number of enumerated URLs exactly matching URLs on the malicious path is identified. Note that we exclude malicious paths that cannot be identified due to there being no Referer header. When no redirection path is matched or multiple redirection paths are identified due to onetime URLs or the same number of matched URLs, we use an approximate matching approach, which was inspired from another approximate matching approach [20]. This approach measures the similarity between URLs by matching the domain name, file path, and query. Finally, we identify the leaf URL on the identified redirection path as a candidate URL because the redirection path corresponds to the malicious path, even though it does not contain any malicious URLs due to evasion. Note that malicious URLs are collected from the detection results of current detection methods such as high-interaction honeyclients.

For example, Fig. 4 depicts two redirection graphs constructed using real-browser traffic (top) and browser-emulator traffic (bottom).
to the malicious path, the identified redirection path is (URL1→URL4→URL6), and our method identifies URL6 as a candidate URL.

3.3 Code Concatenation

The code concatenation component of our method analyzes JavaScript code executed when accessing a candidate URL identified in the previous section. There are various methods of executing JavaScript code, for example, executing code enclosed by a script tag, code loaded from an external URL in a script tag, and dynamically generated code. As mentioned above, attackers prevent our analysis using scattered/obfuscated code in malicious websites. When we identify evasion techniques with subsequent manual analysis, it is inefficient and time consuming to analyze scattered code and each code before and after obfuscation one by one. Hence, our method identifies the data dependencies and dynamic execution relationships between code pieces and concatenates them on the basis of these relationships.

Concatenation based on data dependency involves identifying variable and function names defined with global scope and concatenating two different code pieces that share the same name. For example, JS1 and JS2 in Fig. 5 are concatenated because they share the function called abc(). These data dependencies are identified by converting code to an abstract syntax tree (AST) and parsing it with Esprima [21]. Note that we excluded variable and function names in the curly brackets of function and object definitions to simplify AST parsing and data dependency identification.

Concatenation based on dynamic execution involves concatenating the generator code and generated code when executing dynamically generated code by, for example, eval() and document.write() functions. For example, JS3 and JS3’ in Fig. 5 are concatenated because JS3 executes dynamically generated JS3’. These dynamic executions are identified by monitoring functions related to dynamic executions and outputting JavaScript execution traces, which was inspired by a current monitoring method [7].

Note that we exclude JavaScript code with AST conversion errors, such as nameless function code that can be set for JavaScript events and setTimeout()/setInterval() functions, since we cannot identify data dependencies.

3.4 Code Classification

Most evasive code changes the subsequent execution path using conditional branches and exception handling on the basis of the evasion results[13]. Although we can efficiently identify evasion techniques by leveraging the evasion nature, a large number of manual code analyses are practically infeasible. Hence, the code classification component of our method reduces the number of analyses by code clustering on the basis of code similarity. First, our method converts JavaScript code to a sequence in which statements related to control-flow changes are recorded in the order of appearance. These statements are AST-nodes corresponding to BreakStatement, CatchClause, ConditionalExpression, ContinueStatement, Do-WhileStatement, ForInStatement, ForOfStatement, ForStatement, FunctionDeclaration, FunctionExpression, IfStatement, ReturnStatement, SwitchStatement, ThrowStatement, TryStatement, WhileStatement, and WithStatement when parsing code with Esprima. Next, we define a code similarity as the percentage of the longest common subsequence (LCS) between these sequences. Let S1 and S2 be sequences, \( \text{len}(S_1) \) represent the length of a sequence, and \( \max(\text{len}(S_1), \text{len}(S_2)) \) return the number with the highest value. The code similarity between S1 and S2 is calculated using the following formula.

\[
\text{CodeSimilarity}(S_1, S_2) = \frac{\text{len}(\text{LCS}(S_1, S_2))}{\max(\text{len}(S_1), \text{len}(S_2))} \tag{1}
\]
We used DBSCAN [22], which is a density-based clustering algorithm, for code clustering. It is robust against outliers and can classify points in low-density regions as noise without determining the number of clusters. Therefore, we can exhaustively survey code pieces by analyzing the representative points in each cluster and type of noise. Note that we define the distance between sequences as an absolute value of “1 − CodeSimilarity.”

4. Evaluation

We evaluated the classification results of evasive-code candidates extracted using the proposed method. We also evaluated threats to the validity of the redirection-path matching, code concatenation, and code classification components of the proposed method.

4.1 Experimental Environment and Dataset

In our experiment, we collected HTTP traffic of malicious websites that launch drive-by downloads were preliminarily detected by crawling public URL blacklists [23], [24] with high-interaction honeyclients [5], i.e., real browsers (IE6 or IE8 on Windows XP). The HTTP traffic of a browser emulator was collected by replaying the real-browser traffic and re-analyzing the same malicious websites using HtmlUnit [25] in ② of Fig. 6. The browser emulator emulated the same client environment, i.e., the UserAgent strings and plugin-version values, of the real browsers to distinguish attacks with evasive code from simple environment-dependent attacks with browser-fingerprinting code. To replay the HTTP traffic of malicious websites, we used a replay server that responds with web content matched to a request URL from a client. Note that it responds with web content using the approximate matching approach discussed in Sect. 3.2 when a request URL cannot be matched due to random strings. We collected a dataset of 20,272 HTTP traffic pairs detected from 2012 to 2016, as shown in Table 1. The HTTP traffic pairs, with which our high-interaction honeyclients failed to identify malicious URLs due to complicated exploit code, were preliminarily excluded from this dataset. In addition, we excluded 459 HTTP traffic pairs in which malicious paths could not be identified due to there being no Referer header and 18,647 HTTP traffic pairs in which the browser emulator reached malicious URLs. Finally, the HTTP traffic pairs of 1,166 malicious websites were input to our pipeline. A total of unique 8,467 pieces of JavaScript code were executed throughout these malicious websites. From all code pieces, the redirection-path matching and code concatenation components of our method reduced the number of code pieces to 4,770 and 2,410, respectively, while excluding 26 code pieces due to an Esprima parse error.

Note that we used a single server with a 40-core 2.6-GHz CPU and 256-GB RAM for dataset collection and code clustering. We also parallelized the calculation of the code similarity and set two DBSCAN parameters, eps and minPts, to 0.2 and 3, respectively. These DBSCAN parameters were determined through simple manual verifications of multiple experimental results with a grid search of parameter values.

4.2 Classification of Evasive Code

As a result of classifying the extracted 2,410 code pieces, 57 clusters and a noise cluster were formed. The calculation time was 584 seconds, but this was mainly required to calculate code similarity based on LCS (Eq. (1)). In addition, we manually analyzed one representative point, i.e., code piece, randomly sampled from these 57 clusters. As a result, these clusters were divided into 21 clusters with JavaScript code that caused redirection differences and 36 clusters without it. In this section, we describe these 57 clusters in detail. We explain the noise cluster in the next section.

We classified the 21 clusters with JavaScript code that caused redirection differences into 4 categories: evasion (EV), bug in browser emulator (BG), limitation of browser emulator (LM), and re-analysis failure (RF), as shown in Ta-
Table 2: Classification results of JavaScript code

| Category                  | Cause of redirection difference | # of Clusters | # of Code pieces |
|---------------------------|---------------------------------|---------------|------------------|
| Evasion (EV)              | Use of original object          | 7             | 636†             |
|                           | Difference in array processing  |               |                  |
|                           | Difference in string processing | 1             | 30               |
|                           | Difference in method processing | 2             | 33               |
| Bug in browser emulator (BG) | Bug in DOM processing         | 2             | 12               |
|                           | Bug in JavaScript event handling| 2             | 25               |
| Limitation of browser emulator (LM) | Use of VBScript      | 1             | 3                |
|                           | CVE-2006-0003                  | 4             | 190              |
|                           | CVE-2013-7331                  | 1             | 18               |
| Re-analysis failure (RF)  | Use of DGA-domain              | 1             |                  |

† Multiple evasion techniques are contained in same piece of JavaScript code.

Table 2. JavaScript code in the EV category abused the differences in code-execution results depending on the JavaScript engine used in a browser. We give more details in Sect. 5 since our objective was to identify the code in this evasion category. The BG category contained JavaScript code that triggered bugs in the browser emulator used in our experiment, i.e., HtmlUnit. The code in this category manipulates JavaScript events and DOM APIs. More precisely, the browser emulator was not redirected to URLs because it did not fire the onload event of the injected iframe tag or throw an exception for a parameter error of the appendChild() function. JavaScript code in the LM category was VBScript and exploit code. The HtmlUnit cannot execute them because it does not have a VBScript engine/compiler and does not emulate vulnerabilities. This is a commonly known limitation with browser emulators. The exploit code was not excluded from our dataset during the pre-processing of filtering HTTP traffic pairs because URLs accessed after exploitation were malicious URLs detected with our high-interaction honeyclients. The RF is the last category in Table 2, which contained JavaScript code that redirects to URLs with random domain names. Our browser emulator failed re-analysis because our replay server responded with empty content, i.e., 404 Not Found, when it did not match a request URL due to the domain name being generated with a domain-generation algorithm (DGA).

In comparison, the other 36 clusters contained 1,232 pieces of benign code such as JavaScript libraries, CMS plugins, and the setting code of analytics services. Since our dataset contained the data of compromised and malicious websites that collect access statistics, clusters of benign code were formed.

4.3 Noise Cluster

Low-density-region points, i.e., code pieces, are classified as noise because of the characteristics of DBSCAN. This means that a noise cluster contains rare code in our dataset and has the possibility of containing evasive code not listed in Table 2. Thus, we manually analyzed 224 code pieces in the noise cluster. As a result, we discovered five code pieces in the evasion category. These code pieces were divided into three types: (1) evasive code in Table 2 with different obfuscation approaches, (2) evasive code in Table 2 injected into benign code, i.e., evasive-code injection, and (3) new evasive code not listed in Table 2. The new evasive code was JavaScript code that abuses the difference in parseInt() method processing. We describe in detail this evasive code in Sect. 5.

4.4 Threats to Validity

4.4.1 Existence of Evasive Code in Redirection-Path Matching

Since our method uses real-browser traffic that contains malicious URLs and browser-emulator traffic that does not, the redirection-path matching component of our method can identify at least one cause of the difference, e.g., evasive code and exploit code, in each HTTP traffic pair. However, such JavaScript code with this difference was not discovered from 26 (2.2%) HTTP traffic pairs, as shown in Fig. 7. As a result of manually analyzing these HTTP traffic pairs in detail, we discovered that HtmlUnit and Esprima failed the analysis due to errors. For example, the HtmlUnit errors were the “Illegal Character” error of the JavaScript engine caused by NULL character “\00” and HTTP 3XX redirection loops. In addition to these errors, we discovered a bug in the document.write() function implementation in HtmlUnit [25]. When an argument of the document.write() function contains triple single quotes after a start tag of an HTML element, the element is not evaluated until the next single quote appears, i.e., the previous single quote is closed, even though the element is closed by the end tag. In our experiment, the browser emulator was
not redirected to malicious URLs because it did not execute the `document.write()` function with an argument of the `iframe` tag strings containing triple single quotes. The Esprima error was the syntax error “Invalid left-hand side in assignment” caused by parsing `window.docume()` in the following code snippet. Since the code was not converted to a sequence due to this error, it was not included in our classification results.

```javascript
try {
  window.docume() = fdd1([ ... snipped ... ]);
}
catch (e) {
  [ ... snipped ... ]
}
```

The above threats were derived from errors in our experimental environment. Therefore, we can improve the evasive code analysis capabilities of our environment by fixing them.

4.4.2 Accuracy in Code Concatenation

The threats to the validity of the `code concatenation` component of our method are incorrect code concatenations. Thus, we checked the existence of incorrectly concatenated code pieces that had the possibility of affecting the subsequent manual code analysis. In our experiment, our method concatenated 4,770 code pieces to 2,410 while excluding duplicate code pieces. These 1,270 and 1,755 concatenations were due to data dependencies and dynamic executions, respectively. Concatenation based on data dependencies incorrectly concatenated two code pairs. However, both concatenations were caused by a loop counter variable and did not affect the subsequent analysis. Although this concatenation concatenates code pieces on the basis of only the existence of variable/function names to simplify parsing code, we can improve the accuracy of this concatenation using strict syntax analysis such as identifying the definition-use relationships of variables/functions used in a program-dependence graph [31]. However, this concatenation is suitable for code with less variable/function name collisions, for example, code with obfuscated variable/function names. There were 792 concatenations caused by such obfuscated names. Concatenation on dynamic executions never incorrectly concatenated code pieces because they were concatenated on the basis of the JavaScript trace information of specific functions, such as `eval()`, related to obfuscations. By using this concatenation, we can efficiently identify whether evasive code is contained in the code before obfuscations, i.e., generator code, or after, i.e., generated code.

Note that if our system did not use the code concatenation component, 72 clusters and 341 noises were formed. Without this component, in addition to increasing the number of clusters and noises, we have to manually analyze obfuscated code and many code snippets with unresolved variables due to the scatter code. Therefore, this component can accurately concatenate code pieces and reduce the cost of manual evasive code analysis.

4.4.3 Sequence Selection in Code Classification

The `code classification` component of our method uses sequences in which statements related to control-flow changes are recorded in order of appearance to calculate the code similarity. We confirmed that our method can accurately classify similar code pieces by inspecting points extracted from randomly sampled clusters. In terms of evasion techniques, there was no mixture in the same cluster. However, attackers can evade our classification if they can create evasive code without control-flow (execution-path) changes. For example, attackers can change execution paths on the server side, not on the client side (i.e., JavaScript), by including evasion results in a request URL. Although we could not discover such a code piece in our dataset, different approaches using dynamic code features, such as JavaScript object dump and exception stack trace, are required for clustering them. However, whatever clustering approach is taken, manual analysis to identify evasion techniques is necessary. This is because evasive code basically abuses commonly unknown differences among JavaScript implementations, and sometimes multiple evasion techniques may be contained in the same JavaScript code piece, as stated in the footnote of Table 2.

5. Case Studies

In this section, we describe notable evasive-code examples abusing the differences among JavaScript engines discovered in our experiment. We also report on the behaviors of these examples using IE8, 9, 10, 11, and the latest versions of Firefox and Chrome on Windows 7.

5.1 Use of Original Object

An evasion technique using the original JavaScript object in specific browsers can not only identify the browser families but also evade honeyclient analysis. For example, the code snippet in Fig. 8 defines the variable `ws` with the Firefox-specific JavaScript object `window.sidebar` [26] at line 1. The variable `ws` is assigned to `NaN` (Not a Number) with Firefox or 0 with the other browsers. As a consequence, the access using Firefox is rejected due to the subsequent loop statement.

5.2 Difference in Array Processing

We discovered a difference in array processing. The code snippet in Fig. 8 abuses the `Array.length` property, the property value of which differs depending on the browser [27] at line 2. The variable `len` is assigned to 3 with IE before version 9 or 2 with the other browsers due to the number of commas in the array. Consequently, the number of executions of the following loop statement changes.
5.3 Difference in String Processing

We also discovered a difference in string processing. For example, the code snippet in Fig. 9 stores the comparison result of "\v" to the variable t1 and changes the execution path on the basis of variable t1 from lines 10 to 14. This code snippet abuses the specification that IE before version 9 returns true for the expression "\v" === "\v" [28]. In the case of the code snippet in Fig. 9, browsers except for IE7, 8, and 9, throw an error and stop the execution due to using the object window["dummy"]["error"].

5.4 Differences in Method Processing

The evasion techniques of method processing abuse the differences in function parameters in addition to return results as well as the difference in array processing.

Generally, the setTimeout() function uses two parameters, a function/code to be executed and the time (optional) to wait before executing the function/code [29]. However, the code snippet in Fig. 10 uses the setTimeout() function with one integer argument and can be executed using only IE after version 10 and the latest versions of Firefox and Chrome. Therefore, this evasion technique can identify old IE versions with only the setTimeout() function call. Note that although the real browsers used in our experiment, i.e., IE6 and IE8, cannot execute this evasive code, we were able to discover this code because it was used with different evasive code.

We generally use the parseInt() function with strings as the first parameter and the radix of the strings as the second parameter. However, when we used the number strings that start from “0” with no radix, there were browsers that interpreted the number strings as octal or decimal [30]. For example, the code snippet in Fig. 11 uses the parseInt() function with the argument “0123” for the conditional branch at line 2. Although IEs before version 8 interpreted “0123” as octal and responded with 83, the other browsers interpreted “0123” as decimal and responded with 123. As a consequence, only IEs can execute obfuscated code from lines 3 to 9.

6. Discussion

6.1 Experimental Environment

We obtained the browser-emulator traffic by re-analyzing HTTP transactions of malicious websites preliminarily detected with the real browsers in our experiment. As described in Sect. 4.2, our browser emulator fails to re-analyze websites with DGA-domains. This is a limitation in our experimental architecture. To mitigate this limitation, the emulator needs to directly access websites on the Inter-
net as well as the real browser. Moreover, our method leverages redirection differences to extract evasive-code candidates. Therefore, evasive code in malicious websites without redirections and at the last step of redirections is preliminarily filtered out. However, these websites have already been exposed to us, and it was easy to extract features and detect them with other techniques. On the other hand, the bugs in Table 2 and Sect. 4.4.1 were derived from HtmlUnit, which we used in our experiment. However, we were able to eliminate these bugs in addition to the differences in array processing, method processing, and DOM processing, as shown in Table 2, by patching HtmlUnit so that it can appropriately behave depending on the emulated client environment on the basis of these findings. These differences are derived from differences among JavaScript implementations, i.e., among the Rhino JavaScript engine in HtmlUnit [25] and JavaScript engines in other real browsers. Therefore, these analysis results can also be obtained by other browser emulators with the Rhino JavaScript engine.

6.2 Taking Advantage of Evasive Code as Signature

We investigated whether the patched evasion techniques discussed in the previous section, especially current evasion techniques that abuse the differences in array processing and setTimeout() method processing, can be used as indicators of malicious websites. In this investigation, we improved HtmlUnit so that it can output the detection logs of these evasion techniques and crawled 861,059 websites with the Alexa Top 1M domain names with the patched HtmlUnit. As a result, the evasive code with the array-process difference was detected in 21,065 (2.4%) websites. Most of the code was unintentionally used in these websites because we detected this evasion technique by detecting only the existence of a comma at the end of an array expression, i.e., “[,]”. To accurately use this evasive code as an indicator, we need to detect whether the evasion result, i.e., “[,]” or “[,].length,” is used for changing execution paths. The other evasive code with the setTimeout() method-process difference was detected in 26 (0.003%) websites. We discovered that these all websites were compromised by Fake jQuery injections [33] and injected with the evasive code. From these results, we can use the evasive code with the setTimeout() method-process difference as an indicator for detecting compromised/suspicious websites.

6.3 Excluding Benign JavaScript Code in Malicious Website

More than half of the clusters and noise were classified as benign code without evasion techniques in our evaluation and reduced the efficiency of manual analysis. Thus, we evaluated whether benign clusters and noise can be excluded using JavaScript code executed in the Alexa websites, as discussed in the previous section. In this evaluation, to prevent evasive code exclusion, we used an exact matching of sequences used in our experiment on sequences converted from the Alexa JavaScript code with more than 25 AST-nodes related to control-flow changes. As a result, 6 code pieces in 2 clusters of the other category in Table 2 and 37 types of noise were matched to the Alexa sequences. Since all the matched code pieces were well-known JavaScript libraries, e.g., jQuery, SWFObject, and Modernizr, we were able to automatically exclude them from our dataset. However, we should choose the best approach according to the size of the dataset because it is time consuming to crawl a large number of benign websites and match sequences.

6.4 Root Cause Analysis of Evasive Code

Our method, which leverages redirection differences, can extract evasive code candidates from a large amount of JavaScript code, i.e., 281 candidates (57 clusters and 224 noises) from 8,467 pieces of JavaScript code in our dataset. However, our method cannot automatically identify the root cause of differences, i.e., evasion techniques. To automate the identification of evasion techniques, various data, such as program traces, snapshots, and test cases, are required [32]. In other words, we need to spend enormous effort to identify rare evasion techniques. However, evasive code abuses commonly unknown differences among JavaScript implementations and changes execution paths depending on the evasion result. By leveraging this evasion nature, we can efficiently identify evasion techniques even if through manual analysis using the JavaScript console in each real browser, i.e., IE8–IE11, Firefox, and Chrome. Using the data volume of our experiment, i.e., 57 clusters and 224 types of noise, we were able to identify evasion techniques within several days.

6.5 Timeline Analysis of Evasive Code

Evasion techniques may change from time to time. Thus, we analyzed the evasion techniques discussed in the previous section in a time series to investigate which evasive code researchers/developers need to pay attention to when designing and implementing defense systems in the future. We discovered that attackers started using the evasive code frequently after 2015, as shown in Fig. 12, which shows that only evasive code that abuses the difference in parseInt() method processing (Evasion ID 1) was used in 2012. The other evasive code that abuses the differences in string processing (Evasion ID 2), setTimeout() method processing (Evasion ID 3), and the original object and the difference in array processing (Evasion ID 4) were used after 2015. We can infer that attackers used simple browser-fingerprinting code with a navigator object only to attack vulnerable clients before 2015. After 2015, attackers started using evasive code to prevent automated analysis in addition to identify clients. The evasive code of Evasion ID 3 was observed for several months. We discovered that this code was injected in compromised websites through a mass injection campaign, called “Fake jQuery injections” [33]. A
security vendor’s blog also reported that malicious websites with the evade code of Evasion ID 4 were built by Angler EK [18]. From these findings, we need to design and implement defense systems compatible to at least these evasion techniques that abuse the differences in array processing and setTimeout() method processing.

6.6 Discovering Latest Evasive Code

We mainly discovered evade code related to IE6 and IE8 on Windows XP since these real browsers were used in our experiment. For future work, we will discover new evade code with the latest browsers such as Edge, Firefox, and Chrome.

7. Related Work

7.1 Evasion Detection

Evasion techniques have been extensively discussed in the malware analysis community. Many malware samples attempt to evade detection by identifying an analysis environment, e.g., a sandbox, and stopping the execution of any malicious activities. Lindorfer et al. [34] developed a method for detecting environment-sensitive malware by comparing its behavior in multiple analysis sandboxes. Kirat et al. [35] also developed a method for detecting evade malware by executing the malware on a bare-metal system and comparing its behavior when executed on other emulation and virtualization-based analysis systems. Evasive malware can leverage various types of information, such as processes, filesystems, registries, hardware components, and network configurations, for sandbox detection because it can inspect the client environment after infecting itself [36]. In comparison, the evade code discussed in this paper uses information that JavaScript can collect before drive-by exploitation in redirection chains for honeyclient detection, and the amount of information is less than that of malware. Although the above methods that detect evade malware by comparing execution results between different analysis systems are similar to our method, our method detects evade code in malicious websites by leveraging the drive-by evade nature obtained from limited differences.

Evasion techniques on malicious websites are commonly known, for example, for checking a Referer header, limiting source IP addresses, and using VM-based detection [12]. Kapravelos et al. [37] in particular reviewed high-interaction honeyclients and discussed their security properties. They introduced a number of possible attacks that leverage weaknesses in the design of high-interaction honeyclients such as the detection of hooked functions, memory/registry manipulations, and JavaScript timebomb. However, they especially focused on evasion techniques used for exploitation because high-interaction honeyclients detect the side effects of a successful exploit rather than detecting the exploit itself. In comparison, we mainly focused on evasion techniques against low-interaction honeyclients before exploitation.

Kapravelos et al. [13] proposed a method called “Revolver” related to our method for automatically detecting evade code by identifying code changes designed to evade drive-by-download detectors through constant website inspections. Revolver extracts code differences from web content before and after attack detections and identifies evade code by clustering the code differences on the basis of the similarity. However, it requires two versions of web content with/without evade code, so malicious websites without evade-code changes are out of the scope of Revolver. Although our method also requires two versions of HTTP transactions, one of the HTTP transactions could be generated from the other in our local experimental environment.

7.2 Malicious Website Detection

Over the past few years, many researchers have proposed methods of crawling, analyzing, and detecting malicious websites. Cova et al. presented a method of analyzing and detecting malicious JavaScript code [8]. This method combines anomaly detection using static and dynamic features of JavaScript code with emulation in HtmlUnit to collect these features. Rieck et al. developed a method for automatic detection and prevention of drive-by downloads in a Web proxy [38]. This method blocks malicious websites by statically and dynamically analyzing JavaScript code on-the-fly. Eshete et al. developed a holistic method for analyzing and detecting malicious websites by leveraging HTML, JavaScript, URL and social-reputation features [10]. These features are extracted through lightweight dynamic analysis with customized HtmlUnit. Moreover, a hybrid method for malicious JavaScript detection and classification has been proposed [11]. This classification can not only explain attack behavior but also potentially discover new malicious JavaScript variants and new vulnerabilities. In addition to the above methods with dynamic JavaScript analysis, re-
searchers have proposed lightweight detection methods with only static JavaScript analysis. Curtlinger et al. leveraged JavaScript features, such as shellcode concatenations with loop statements and exploitations with exception handling, and expressed these features by using AST contextual information to detect malicious JavaScript [39]. Canali et al. proposed a method for quickly filtering out benign websites and extracting only suspicious webpages for the subsequent costly analyzers using 70 HTML, JavaScript, and URL features [40].

Many methods of detecting a redirection graph on malicious websites rather than malicious web content have also been proposed. Graph-based methods using the behavioral information of browsers construct a redirection graph on the basis of redirection information collected from a number of honeypots or a user’s clients [41], [42]. These methods detect malicious websites by leveraging co-occurring URLs in graphs and a diverse dataset of graphs.

All the above methods are not for detecting evasive code but for detecting malicious URLs, exploit code, and malware after evasion. Those using a browser emulator cannot obtain enough information to detect malicious websites due to the evasive code discovered in this study. Therefore, our method complements the above methods, and our findings will contribute to further improving the analysis capabilities of these conventional methods.

8. Conclusion

We proposed a method of extracting evasive-code candidates through a differential analysis of HTTP transactions obtained using a real browser and browser emulator. As a result of analyzing 8,467 JavaScript samples executed in 20,272 malicious websites with our method, we discovered previously unknown evasion techniques abusing the differences among JavaScript implementations and preventing browser emulator analysis. We will need to design and implement countermeasure techniques considering these evasion techniques to fight malicious websites in the future.

References

[1] Y. Takata, M. Akiyama, T. Yagi, T. Hariu, and S. Goto, “Understanding Evasion Techniques that Abuse Differences among JavaScript Implementations,” Proc. International Conference on Web Information Systems Engineering (WISE), vol.10570, pp.278–294, 2017.
[2] N. Provos, P. Mavrommatis, M.A. Rajab, and F. Monrose, “All your iframes point to us,” Proc. USENIX Security Symposium, 2008.
[3] The HoneyeIR Project, “Capture-HPC,” https://projects.honeyeir.org/capture-hpc
[4] L. Liu, V. Vengneswaran, P. Porras, and W. Lee, “BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections,” Proc. ACM Conference on Computer and Communications Security (CCS), 2010.
[5] M. Akiyama, T. Yagi, Y. Kadobayashi, T. Hariu, and S. Yamaguchi, “Client Honeypot Multiplication with High Performance and Precise Detection,” IEICE Trans. Inf. & Syst., vol.E98-D, no.4, pp.775–787, 2015.
[6] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, “Rozzle: Decloaking Internet Malware,” Proc. IEEE Symposium on Security and Privacy (SP), pp.443–457, 2012.
[7] Y. Takata, M. Akiyama, T. Yagi, T. Yada, and S. Goto, “Website Forensic Investigation to Identify Evidence and Impact of Compromise,” Proc. International Conference on Security and Privacy in Communication Networks (SecureComm), 2016.
[8] M. Cova, C. Kruegel, and G. Vigna, “Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code,” Proc. World Wide Web Conference (WWW), p.281, 2010.
[9] A. Dell’ Aera, “Thug,” http://buffer.github.io/thug/
[10] B. Eshete, A. Villaforita, and K. Wellemerami, “BINSPECT: Holistic Analysis and Detection,” Proc. International Conference on Security and Privacy in Communication Networks (SecureComm), 2013.
[11] J. Wang, Y. Xue, Y. Liu, and T.H. Tan, “JSICD: A Hybrid Approach for JavaScript Malware Detection and Classification,” Proc. ACM Symposium on Information, Computer and Communications Security (AsiaCCS), 2015.
[12] M.A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt, “Trends in Circumventing Web-Malware Detection,” Google Technical Report, July 2011.
[13] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, “Revolver: An Automated Approach to the Detection of Evasive Web-based Malware,” Proc. USENIX Security Symposium, 2013.
[14] C. Grier, L. Ballard, J. Caballero, N. Chachra, C.J. Dietrich, K. Lewchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillides, N. Provos, M.Z. Rafique, M.A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G.M. Voelker, “Manufacturing Compromise: The Emergence of Exploit-as-a-Service,” Proc. ACM Conference on Computer and Communications Security (CCS), 2012.
[15] B. Stock, B. Livshits, and B. Zorn, “Kizzle: A Signature Compiler for Exploit Kits,” Proc. International Conference on Dependable Systems and Networks (DSN), 2016.
[16] G. Lu and S. Debray, “Weaknesses in Defenses against Web-Borne Malware,” Proc. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), vol.7967, pp.139–149, 2013.
[17] D.Y. Wang, S. Savage, and G.M. Voelker, “Cloak and dagger: dynamics of web search cloaking,” Proc. ACM Conference on Computer and Communications Security (CCS), 2011.
[18] Y. Zhou and W. Xu, “Angler Exploit Kit Continues to Evade Detection: Over 90,000 Websites Compromised,” http://researchcenter.paloaltonetworks.com/2016/01/angler-exploit-kit-continues-to-evade-detection-over-90000-websites-compromised/
[19] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahmad, “WebWitness: Investigating, Categorizing, and Mitigating Malware Download Paths,” Proc. USENIX Security Symposium, 2015.
[20] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms, “ClickMiner: Towards Forensic Reconstruction of User-Browser Interactions from Network Traces Categories and Subject Descriptors,” Proc. ACM Conference on Computer and Communications Security (CCS), 2014.
[21] A. Hidayat, “Esprima,” http://esprima.org/
[22] M. Estor, H.-P. Krieger, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” Proc. International Conference on Knowledge Discovery and Data Mining (KDD), 1996.
[23] “Malware Domain List,” https://www.malwaredomainlist.com/
[24] “Malwarebytes, “hpHosts,” https://www.hosts-file.net/
[25] Gargoyle Software Inc., “HtmlUnit,” http://htmlunit.sourceforge.net/
[26] Mozilla Developer Network, “window.setTimeout,” https://developer.mozilla.org/en-US/docs/Web/API/window.setTimeout
[27] stack overflow, “Javascript Browser Quirks - array.Length,” http://stackoverflow.com/questions/29053/javascript-browser-quirks-array-length
[28] Microsoft Developer Network, “Special Characters (JavaScript),” https://msdn.microsoft.com/en-us/library/2yf77377(v=vs.94).aspx
[29] Mozilla Developer Network, “window.setTimeout,” https://developer.mozilla.org/en-US/docs/Web/API/Window/Timeout/setTimeout
Mozilla Developer Network, “parseInt.” https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt

J. Ferrante, K.J. Ottenstein, and J.D. Warren, “The Program Dependence Graph and Its Use in Optimization,” ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319–349, 1987.

P. Saxena, D. Akhawe, and S. Hanna, “A symbolic execution framework for javascript,” Proc. IEEE Symposium on Security and Privacy (SP), pp. 513–528, 2010.

D. Sinegubko, “jQuery.min.php Malware Affects Thousands of Websites,” https://blog.sucuri.net/2015/11/jquery-min-php-malware-affects-thousands-of-websites.html

M. Lindorfer, C. Kolbitsch, and F.M. Comparetti, “Detecting environment-sensitive malware,” Proc. Research in Attacks, Intrusions and Defense (RAID), 2011.

D. Kirat, G. Vigna, and C. Kruegel, “BareCloud: Bare-metal Analysis-based Evasive Malware Detection,” Proc. USENIX Security Symposium, 2014.

A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes, and C. Rossow, “SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion,” Proc. Research in Attacks, Intrusions and Defense (RAID), vol. 9854, pp. 165–187, 2016.

A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna, “Escape from Monkey Island: Evading High-Interaction Honeyclients,” Proc. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), vol. 6739, pp. 124–143, 2011.

K. Ricke, T. Krueger, and A. Dewald, “Cujo: efficient detection and prevention of drive-by-download attacks,” Proc. Annual Computer Security Applications Conference (ACSAC), 2010.

C. Curtisinger, B. Livshits, B. Zorn, and C. Seifert, “ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection,” Proc. USENIX Security Symposium, 2011.

D. Canali, M. Cova, and G. Vigna, “Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages Categories and Subject Descriptors,” Proc. World Wide Web Conference (WWW), 2011.

J. Zhang, C. Seifert, J.W. Stokes, and W. Lee, “Arrow: Generating Signatures to Detect Drive-by-Downloads,” Proc. International World Wide Web Conference (WWW), 2011.

G. Stringhini, C. Kruegel, and G. Vigna, “Shady Paths: Leveraging Surfing Crowds to Detect Malicious Web Pages Categories and Subject Descriptors,” Proc. ACM Conference on Computer and Communications Security (CCS), 2013.

Yuta Takata received his B.E., M.E., and Ph.D. degrees in computer science and engineering from Waseda University, Japan in 2011, 2013, and 2018. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2013, he has been engaged in research and development on cyber security using honeypot/honeyclient and program/content analysis techniques.

Mitsuaki Akiyama received his M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology, Japan in 2007 and 2013. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2007, he has been engaged in research and development on network security, especially honeypot and malware analysis. He is now with the Cyber Security Project of NTT Secure Platform Laboratories.

Takeshi Yagi received his B.E. degree in electrical and electronic engineering and his M.E. degree in science and technology from Chiba University, Japan in 2000 and 2002. He also received his Ph.D. degree in information science and technology from Osaka University, Osaka, Japan in 2013. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2002, he has been engaged in research and design on network architecture, traffic engineering, and his current research interests include network security, web security, and cyber security. He is now a senior research engineer in the Cyber Security Project of NTT Secure Platform Laboratories. He is a member of the Institute of Electrical and Electronics Engineers (IEEE) and the Institute of Electrical Engineers of Japan (IEEE).

Takeo Hariu is a senior research engineer, and supervisor at Nippon Telegraph and Telephone Corporation (NTT). He received his M.S. degree in electro-communications from the University of Electro-Communications. Contact him at hariu.takeo@lab.ntt.co.jp.

Kazuhiko Ohkubo is a vice president and the head of NTT Secure Platform Laboratories. He received his B.S. in information engineering from the University of Tsukuba in 1987 and M.S. in electrical engineering from the University of Tokyo in 1989. He also earned his M.S. degree in management of technology from the MIT Sloan School of Management, USA in 2000. He is a member of IEICE and IEEE, and he is an outside director of NTT Security Corporation, the specialized security company of NTT

Shigeki Goto is a professor in the Department of Computer Science and Engineering, Waseda University, Japan. He received his B.S. and M.S. in mathematics from the University of Tokyo. Prior to becoming a professor at Waseda University, he worked for NTT for many years. He also earned his Ph.D. in information engineering from the University of Tokyo. He is the president of JPNIC. He is a member of ACM and IEEE, and he was a trustee of the Internet Society from 1994 to 1997.