Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code

Marco Cova, Christopher Kruegel, and Giovanni Vigna
University of California, Santa Barbara
{marco,chris,vigna}@cs.ucsb.edu

ABSTRACT

JavaScript is a browser scripting language that allows developers to create sophisticated client-side interfaces for web applications. However, JavaScript code is also used to carry out attacks against the user's browser and its extensions. These attacks usually result in the download of additional malware that takes complete control of the victim's platform, and are, therefore, called "drive-by downloads." Unfortunately, the dynamic nature of the JavaScript language and its tight integration with the browser make it difficult to detect and block malicious JavaScript code.

This paper presents a novel approach to the detection and analysis of malicious JavaScript code. Our approach combines anomaly detection with emulation to automatically identify malicious JavaScript code and to support its analysis. We developed a system that uses a number of features and machine-learning techniques to establish the characteristics of normal JavaScript code. Then, during detection, the system is able to identify anomalous JavaScript code by emulating its behavior and comparing it to the established profiles. In addition to identifying malicious code, the system is able to support the analysis of obfuscated code and to generate detection signatures for signature-based systems. The system has been made publicly available and has been used by thousands of analysts.

Categories and Subject Descriptors

K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security and Protection*

General Terms

Security

Keywords

Drive-by-download attacks, web client exploits, anomaly detection

1. INTRODUCTION

Malicious web content has become the primary instrument used by miscreants to perform their attacks on the Internet. In particular, attacks that target web clients, as opposed to infrastructure components, have become pervasive [28].

Drive-by downloads are a particularly common and insidious form of such attacks [29]. In a drive-by download, a victim is lured to a malicious web page. The page contains code, typically written in the JavaScript language, that exploits vulnerabilities in the user's browser or in the browser's plugins. If successful, the exploit downloads malware on the victim machine, which, as a consequence, often becomes a member of a botnet.

Several factors have contributed to making drive-by-download attacks very effective. First, vulnerabilities in web clients are widespread (in 2008, such vulnerabilities constituted almost 15% of the reports in the CVE repository [18]), and vulnerable web clients are commonly used (about 45% of Internet users use an outdated browser [8]). Second, attack techniques to reliably exploit web client vulnerabilities are well-documented [4, 33–35]. Third, sophisticated tools for automating the process of fingerprinting the user's browser, obfuscating the exploit code, and delivering it to the victim, are easily obtainable (e.g., NeoSploit, and LuckySploit [15]).

The mix of widespread, vulnerable targets and effective attack mechanisms has made drive-by downloads the technique of choice to compromise large numbers of end-user machines. In 2007, Provos et al. [28] found more than three million URLs that launched drive-by-download attacks. Even more troubling, malicious URLs are found both on rogue web sites, that are set up explicitly for the purpose of attacking unsuspecting users, and on legitimate web sites, that have been compromised or modified to serve the malicious content (high-profile examples include the Department of Homeland Security and the BusinessWeek news outlet [10, 11]).

A number of approaches have been proposed to detect malicious web pages. Traditional anti-virus tools use static signatures to match patterns that are commonly found in malicious scripts [2]. Unfortunately, the effectiveness of syntactic signatures is thwarted by the use of sophisticated obfuscation techniques that often hide the exploit code contained in malicious pages. Another approach is based on low-interaction honeyclients, which simulate a regular browser and rely on specifications to match the behavior, rather than the syntactic features, of malicious scripts (for example, invoking a method of an ActiveX control vulnerable to buffer overflows with a parameter longer than a certain length) [14, 23]. A problem with low-interaction honeyclients is that they are limited by the coverage of their specification database; that is, attacks for which a specification is not available cannot be detected. Finally, the state-of-the-art in malicious JavaScript detection is represented by high-interaction honeyclients. These tools consist of full-featured web browsers typically running in a virtual machine. They work by monitoring all modifications to the system environment, such as files created or deleted, and processes launched [21, 28, 37, 39]. If any unexpected modification occurs, this is considered as the manifestation of an attack, and the corresponding page is flagged as malicious. Unfortunately, also high-interaction honeyclients have limitations. In particular, an attack can be detected only if the vulnerable component (e.g., an ActiveX control or a browser plugin) targeted by the exploit is installed and correctly activated on the de-

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

tection system. Since there exist potentially hundreds of such vulnerable components, working under specific combinations of operating system and browser versions, the setup of a high-interaction honeyclient and its configuration is difficult and at risk of being incomplete. As a consequence, a significant fraction of attacks may go undetected. (Indeed, Seifert, the lead developer of a popular high-interaction honeyclient, says, "high-interaction client honeypots have a tendency to fail at identifying malicious web pages, producing false negatives that are rooted in the detection mechanism" [32].)

In this paper, we propose a novel approach to the automatic detection and analysis of malicious web pages. For this, we visit web pages with an instrumented browser and record events that occur during the interpretation of HTML elements and the execution of JavaScript code. For each event (e.g., the instantiation of an ActiveX control via JavaScript code or the retrieval of an external resource via an `iframe` tag), we extract one or more features whose values are evaluated using anomaly detection techniques. Anomalous features allow us to identify malicious content even in the case of previously-unseen attacks. Our features are comprehensive and model many properties that capture intrinsic characteristics of attacks. Moreover, our system provides additional details about the attack. For example, it identifies the exploits that are used and the unobfuscated version of the code, which are helpful to explain how the attack was executed and for performing additional analysis.

We implemented our approach in a tool called JSAND (JavaScript Anomaly-based aNalysis and Detection), and validated it on over 140,000 web pages. In our experiments, we found that our tool performed significantly better than existing approaches, detecting more attacks and raising a low number of false positives. We also made JSAND available as part of an online service called Wepawet (at `http://wepawet.cs.ucsb.edu`), where users can submit URLs and files that are automatically analyzed, delivering detailed reports about the type of observed attacks and the targeted vulnerabilities. This service has been operative since November 2008 and analyzes about 1,000 URLs per day submitted from users across the world.

In summary, our main contributions include:

- A novel approach that has the ability to detect previously-unseen drive-by downloads by using machine learning and anomaly detection.
- The identification of a set of ten features that characterize intrinsic events of a drive-by download and allow our system to robustly identify web pages containing malicious code.
- An analysis technique that automatically produces the deobfuscated version of malicious JavaScript code, characterizes the exploits contained in the code, and generates exploit signatures for signature-based tools.
- An online service that offers public access to our tool.

2. BACKGROUND

Before introducing our detection approach, we briefly describe a drive-by-download exploit that we found in the wild.

Redirections, Fingerprinting, and Obfuscations. The attack was triggered when visiting a benign web site that has fallen victim to a SQL injection attack. As a consequence, the HTML code `<script src="http://www.kjwd.ru/js.js">` was injected into all pages of the vulnerable site. The injected code causes a script from `www.kjwd.ru` to be executed. This script checks if the visitors has been attacked recently (in which case, a cookie with a specific name would have been set). If no previous attack is detected, the script sets the cookie to mark the beginning of the attack, and injects an `iframe` tag into the web page. This iframe is declared with null height and width, so it is invisible to the user.

The injected iframe points to a resource hosted on a third web site, namely `iroe.ru`. This web site uses the `User-Agent` request-header field to detect the user's browser and operating system. Depending on the detected versions, a different page is returned to the browser. In this example, our client presents itself as Internet Explorer 6.1 running on Windows XP. As a result, the returned page consists of a JavaScript document that carries out exploits against this browser and platform. Different combinations of browser and OS brand and version may lead to different exploits being served, or to a completely benign behavior.

The returned script uses various techniques to make its analysis more complex. For example, it uses polymorphism and its variable and function names are generated randomly each time the script is requested. It is also obfuscated and most of its code is dynamically generated and executed (using the JavaScript `eval` function). The code also uses various techniques to thwart simple analysis. For example, it uses the value of the property `location.href` as a key to decode encrypted strings. As a consequence, an off-line analysis of the script will fail, since it changes the location to an incorrect value.

Exploits. After the deobfuscation step, the actual malicious code is revealed and ready to execute.

```
1 function a9_bwCED() {
2   var OBGUiGAa = new ActiveXObject('Sb.SuperBuddy');
3   if (OBGUiGAa) {
4     Exhne69P();
5     OBGUiGAa.LinkSBIcons(0x0c0c0c0c);
6   }
7   return 0;
8 }
9 if (a9_bwCED() || g0UnHabs() || P9i182jC())  { ... }
```

The code attempts to execute three exploits (line 9) targeting different vulnerabilities. We will describe the attack launched by the `a9_bwCED` function (the other two attacks are similar). First, the code attempts to instantiate the vulnerable component (in this case, the SuperBuddy control, at line 2). If it can be instantiated, `Exhne69P` is invoked (line 4). This function loads the shellcode into the heap, and, by allocating many carefully-chosen strings through substring and concatenation operations, it controls the heap layout so that the shellcode is very likely to be reached if the program's control is hijacked by the exploit. This is a technique called "heap spraying" [33]. Finally, the actual exploit is triggered, in this case, by invoking a specific method of the ActiveX control with a large integer, causing an integer overflow (line 5).

In the last step of a successful exploit, the injected shellcode is executed. The shellcode usually downloads malware from a web site and executes it. As a consequence, the compromised machine typically becomes part of a botnet [26].

Discussion. It is interesting to observe how the techniques used by current malicious JavaScript code affect the applicability and effectiveness of different approaches to its detection and analysis. First, malicious JavaScript code frequently uses techniques such as polymorphism, obfuscation, and encoding, which effectively thwart purely syntactic approaches, such as static signatures. In addition, malicious code extensively relies on the dynamic features offered by the JavaScript language, e.g., run-time code generation and evaluation. These constructs complicate the use of static analysis approaches, which cannot generally model with sufficient precision dynamic language features. Furthermore, the attacks target vulnerabilities in the browser itself and a large number of third-party components and applications. Tools that rely on observing the effects of a successful attack require an extensive configuration pro-

cess to install these additional components. This is undesirable, as more resources are required to setup and maintain the detection systems. Finally, since new vulnerabilities are discovered frequently and quickly become exploited in the wild, databases of known exploits or known vulnerable components are also inadequate.

3. DETECTION APPROACH

We have seen that sophisticated JavaScript-based malware is difficult to detect and analyze using existing approaches. Thus, there is the need for a novel approach that overcomes current challenges. This approach has to be robust to obfuscation techniques, must handle accurately the dynamic features of JavaScript, and should not require reconfiguration when new vulnerabilities are exploited. To do so, our approach relies on comprehensive dynamic analysis and anomaly detection.

3.1 Features

Anomaly detection is based on the hypothesis that malicious activity manifests itself through anomalous system events [5]. Anomaly detection systems monitor events occurring in the system under analysis. For each event, a number of features are extracted. During a learning phase, "normal" feature values are learned, using one or more models. After this initial phase, the system is switched to detection mode. In this mode, the feature values of occurring events are assessed with respect to the trained models. Events that are too distant from the established models of normality are flagged as malicious.

In our system, the features characterize the events (e.g., the instantiation of an ActiveX control, the invocation of a plugin's method, or the evaluation of a string using the `eval` function) occurring during the interpretation of the JavaScript and HTML code of a page. In the following, we introduce the features used in our system by following the steps that are often followed in carrying out an attack, namely *redirection and cloaking*, *deobfuscation*, *environment preparation*, and *exploitation*.

3.1.1 Redirection and cloaking

Typically, before a victim is served the exploit code, several activities take place. First, the victim is often sent through a long chain of redirection operations. These redirections make it more difficult to track down an attack, notify all the involved parties (e.g., registrars and providers), and, ultimately, take down the offending sites.

In addition, during some of these intermediate steps, the user's browser is fingerprinted. Depending on the obtained values, e.g., brand, version, and installed plugins, extremely different scripts may be served to the visitor. These scripts may be targeting different vulnerabilities, or may redirect the user to a benign page, in case no vulnerability is found.

Finally, it is common for exploit toolkits to store the IP addresses of victims for a certain interval of time, during which successive visits do not result in an attack, but, for example, in a redirection to a legitimate web site.

We monitor two features that characterize this kind of activity:

Feature 1: *Number and target of redirections*. We record the number of times the browser is redirected to a different URI, for example, by responses with HTTP Status 302 or by the setting of specific JavaScript properties, e.g., `document.location`. We also keep track of the targets of each redirection, to identify redirect chains that involve an unusually-large number of domains.

Feature 2: *Browser personality and history-based differences*. We visit each resource twice, each time configuring our browser with a different personality, i.e., type and version. For example, on the first visit, we announce the use of Internet Explorer, while, on the second, we claim to be using Firefox. The visits are originated from the same IP address. We then measure if the returned pages differ in terms of their network and exploitation behavior. For this, we define the distance between the returned pages as the number of different redirections triggered during the visits and the number of different ActiveX controls and plugins instantiated by the pages.

An attacker could evade these features by directly exposing the exploit code in the target page and by always returning the same page and same exploits irrespective of the targeted browser and victim. However, these countermeasures would make the attack less effective (exploits may target plugins that are not installed on the victim's machine) and significantly easier to track down.

3.1.2 Deobfuscation

Most of the malicious JavaScript content is heavily obfuscated. In fact, it is not rare for these scripts to be hidden under several layers of obfuscation. We found that malicious scripts use a large variety of specific obfuscation techniques, from simple encodings in standard formats (e.g., base64) to full-blown encryption. However, all techniques typically rely on the same primitive JavaScript operations, i.e., the transformations that are applied to an encoded string to recover the clear-text version of the code. In addition, deobfuscation techniques commonly resort to dynamic code generation and execution to hide their real purpose.

During execution, we extract three features that are indicative of the basic operations performed during the deobfuscation step:

Feature 3: *Ratio of string definitions and string uses*. We measure the number of invocations of JavaScript functions that can be used to define new strings (such as `substring`, and `fromCharCode`), and the number of string uses (such as `write` operations and `eval` calls). We found that a high def-to-use ratio of string variables is often a manifestation of techniques commonly used in deobfuscation routines.

Feature 4: *Number of dynamic code executions*. We measure the number of function calls that are used to dynamically interpret JavaScript code (e.g., `eval` and `setTimeout`), and the number of DOM changes that may lead to executions (e.g., `document.write`, `document.createElement`).

Feature 5: *Length of dynamically evaluated code*. We measure the length of strings passed as arguments to the `eval` function. It is common for malicious scripts to dynamically evaluate complex code using the `eval` function. In fact, the dynamically evaluated code is often several kilobytes long.

An attacker could evade these features by not using obfuscation or by devising obfuscation techniques that "blend" with the behavior of normal pages, in a form of mimicry attack [38]. This would leave the malicious code in the clear, or would significantly constrain the techniques usable for obfuscation. In both cases, the malicious code would be exposed to simple, signature-based detectors and easy analysis.

3.1.3 Environment preparation

Most of the exploits target memory corruption vulnerabilities. In these cases, the attack consists of two steps. First, the attacker injects into the memory of the browser process the code she wants to execute (i.e., the *shellcode*). This is done though legitimate operations, e.g., by initializing a string variable in a JavaScript program with the shellcode bytes. Second, the attacker attempts to hijack the browser's execution and direct it to the shellcode. This step is done by exploiting a vulnerability in the browser or one of its components, for example, by overwriting a function pointer through a heap overflow.

One problem for the attacker is to guess a proper address to jump to. If the browser process is forced to access a memory address that does not contain shellcode, it is likely to crash, causing the attack to fail. In other words, reliable exploitation requires that the attacker have precise control over the browser's memory layout. A number of techniques to control the memory layout of browsers have been recently proposed [4, 33, 35]. Most of these techniques are based on the idea of carefully creating a number of JavaScript strings. This will result in a series of memory allocations and deallocations in the heap, which, in turn, will make it possible to predict where some of the data, especially the shellcode, will be mapped.
To model the preparatory steps for a successful exploit, we extract the following two features:

Feature 6: *Number of bytes allocated through string operations*. String functions, such as assignments, `concat`, and `substring` are monitored at run-time to keep track of the allocated memory space. Most techniques employed to engineer reliable heap exploits allocate a large amount of memory. For example, exploits using the heap spraying technique commonly allocate in excess of 100MB of data.

Feature 7: *Number of likely shellcode strings*. Exploits that target memory violation vulnerabilities attempt to execute shellcode. Shellcode can be statically embedded in the text of the script, or it can be dynamically created. To identify static shellcode, we parse the script and extract strings longer than a certain threshold (currently, 256 bytes) that, when interpreted as Unicode-encoded strings, contain non-printable characters. Similar tests on the length, encoding, and content type are also performed on strings created at run-time.

Fine-grained control of the memory content is a necessary requirement for attacks that target memory corruption errors (e.g., heap overflows or function pointer overwrites). While improvements have been proposed in this area [34], the actions performed to correctly set up the memory layout appear distinctively in these features. The presence of shellcode in memory is also required for successful memory exploits.

3.1.4 Exploitation

The last step of the attack is the actual exploit. Since the vast majority of exploits target vulnerabilities in ActiveX or other browser plugins, we extract the following three features related to these components:

Feature 8: *Number of instantiated components*. We track the number and type of browser components (i.e., plugins and ActiveX controls) that are instantiated in a page. To maximize their success rate, exploit scripts often target a number of vulnerabilities in different components. This results in pages that load a variety of unrelated plugins or that load the same plugin multiple times (to attempt an exploit multiple times).

Feature 9: *Values of attributes and parameters in method calls*. For each instantiated component, we keep track of the values passed as parameters to its methods and the values assigned to its properties. The values used in exploits are often very long strings, which are used to overflow a buffer or other memory structures, or large integers, which represent the expected address of the shellcode.

Feature 10: *Sequences of method calls*. We also monitor the sequences of method invocations on instantiated plugins and ActiveX controls. Certain exploits, in fact, perform method calls that are perfectly normal when considered in isolation, but are anomalous (and malicious) when combined. For example, certain plugins allow to download a file on the local machine and to run an executable from the local file system. An attack would combine the two calls to download malware and execute it.

The exploitation step is required to perform the attack and compromise vulnerable components. Of course, different types of attacks might affect certain features more than others. We found that, in practice, these three features are effective at characterizing a wide range of exploits.

3.1.5 Feature Robustness and Evasion

The ten features we use characterize the entire life cycle of an exploit, from the initial request to the actual exploitation of vulnerable components. This gives us a comprehensive picture of the behavior of a page.

We observe that our ten features can be classified into two categories: *necessary* and *useful*. Necessary features characterize actions that are required for a successful exploit. These include the environment preparation features (Feature 6 and 7) and the exploitation features (Feature 8, 9, and 10). Useful features characterize behaviors that are not strictly required to launch a successful attack, but that allow attackers to hide malicious code from detectors and to make it more difficult to track and shut down the involved web sites. These are the redirection and cloaking features (Feature 1 and 2) and the deobfuscation features (Feature 3, 4, and 5).

We claim that our feature set is difficult to evade. To support this claim, we examined hundreds of publicly available exploit scripts and vulnerability descriptions. We found that attacks target three general classes of vulnerabilities: plugin memory violations (e.g., overflows in plugins or ActiveX components), unsafe APIs (e.g., APIs that, by design, allow to perform unsafe actions, such as downloading and executing a remote file), and browser memory violations (e.g., overflows in some of the core browser components, such as its XML parser). Table 1 shows that, for each of these three vulnerability classes, at least two necessary features characterize the actions that are required to successfully perform an exploit. In fact, memory violations require to inject a shellcode in the browser's memory (Feature 7) and to properly set up the memory layout (Feature 6). Overflows require the use of anomalous parameters, such as long strings or large integer (Feature 9). Anomalous parameters or sequences of method calls are necessary in the exploitation of unsafe APIs, for example the use of the path to an executable where usually the path to an HTML file is found (Feature 10). Finally, plugin-related vulnerabilities require loading the vulnerable plugin or ActiveX control (Feature 8). This shows that the feature set we identified is robust against evasion.

Finally, it is possible that benign pages display some of the behaviors that we associate with drive-by-download attacks. For example, fingerprinting of the user's browser can be done to compute access statistics, long redirection chains are typical in syndicated ad-networks, and differences in a page over multiple visits may be caused by different ads being displayed. However, we argue that it is unlikely for a page to have a behavior that matches a combination of our features (especially, the necessary features). We will examine this issue more in detail when measuring the false positives of our approach.

3.2 Models

In the context of anomaly detection, a model is a set of procedures used to evaluate a certain feature. More precisely, the task of a model is to assign a probability score to a feature value. This probability reflects the likelihood that a given feature value occurs, given an established model of "normality." The assumption is that feature values with a sufficiently low probability are indication of a potential attack.

A model can operate in training or detection mode. In training mode, a model learns the characteristics of normal events and deter-

Attack Class	Example Vulnerability	Useful Features					Necessary Features				
		F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Plugin memory violation	CVE-2008-1027						X	X	X	X	
Plugin unsafe API	CVE-2006-0003								X	X	X
Browser memory violation	CVE-2008-4844						X	X			

Table 1: Required and optional features for each attack class. An `X` in a column means that the corresponding feature characterizes a required step in an attack. Features are numbered from 1 to 10 as in Section 3.

mines the threshold to distinguish between normal and anomalous feature values. In detection mode, the established models are used to determine an anomaly score for each observed feature value. For our system, we use several models provided by libAnomaly, a library to develop anomaly detection systems [16,17]. Here, we only briefly describe these models and refer the interested reader to the original references for further information.

Token Finder. The Token Finder model determines if the values of a certain feature are elements of an enumeration, i.e., are drawn from a limited set of alternatives. In legitimate scripts, certain features can often have a few possible values. For example, in an ActiveX method that expects a Boolean argument, the argument values should always be 0 or 1. If a script invokes that method with a value of `0x0c0c0c0c`, the call should be flagged as anomalous.

We apply this model to each method parameter and property value exposed by plugins.

String Length and Occurrence Counting. The goal of this model is to characterize the "normal" length of a string feature. We also use it to model the expected range of a feature that counts the occurrence of a certain event. The rationale behind this model is that, in benign scripts, strings are often short and events occur only a limited number of times. During an attack, instead, longer strings are used, e.g., to cause overflows, and certain events are repeated a large number of times, e.g., memory allocations used to set up the process heap.

We use this model to characterize the length of string parameters passed to methods and properties of plugins, and the length of dynamically evaluated code. In addition, we use it to characterize all the features that count how many times a certain event repeats, i.e., the number of observed redirections, the ratio of string definitions and uses, the number of code executions, the number of bytes allocated through string operations, the number of likely shellcode strings, and the number of instantiated plugins.

Character Distribution. The Character Distribution model characterizes the expected frequency distribution of the characters in a string. The use of this model is motivated by the observation that, in most cases, strings used in JavaScript code are human-readable and are taken from a subset of some well-defined character set. On the contrary, attacks often employ strings containing unusual characters, e.g., non-printable characters, in order to encode binary code or to represent memory addresses.

We use this model to characterize the values passed as arguments to methods and to properties of plugins.

Type Learner. The Type Learner model was not present in the original libAnomaly library, and we added it for this work. This model determines if the values of a certain feature are always of the same type. For example, during normal usage, the parameter of a plugin's method may always contain a URL. However, during an attack, the parameter may be used to overwrite a function pointer with a specific memory address (i.e., a large integer). In this case, the Type Learner would flag this value as anomalous.

In training mode, the Type Learner classifies feature values as one of several possible types. The types currently recognized are small integers (integer values smaller or equal to 1024), large integers, strings containing only printable characters, URLs, and strings identifying a path to an executable file. If all values observed during the training phase are classified as the same type, that type is inferred for the feature. If the type inference fails or is inconsistent, no type is assigned to the corresponding feature. In detection mode, if a type was determined for a feature and the observed value is of that same type, the value is considered normal. Otherwise, it is flagged as anomalous. If no type could be determined, the model always reports a normal score.

The scores of all models are combined in a weighted sum to form the overall anomaly score of a web page. Currently, we assign the same weight to each model. If the overall score is above the threshold established during training, the page is flagged as malicious.

3.3 Emulation

To deal with exploits that heavily rely on dynamic JavaScript features and sophisticated browser functionality, we visit web pages with a customized browser, which loads the page, executes its dynamic content, and records the events used by the anomaly detection system. In particular, in our system, a full browser environment is emulated by HtmlUnit, a Java-based framework for testing web-based applications [9]. HtmlUnit models HTML documents and provides an API to interact with these documents. It supports JavaScript by integrating Mozilla's Rhino interpreter [22]. HtmlUnit implements the standard functionality provided by regular browsers, except visual page rendering. We have instrumented HtmlUnit and Rhino to extract the features used to detect and analyze malicious code.

We have decided to use HtmlUnit rather than instrumenting a traditional browser, such as Firefox or Internet Explorer, for several reasons. First, HtmlUnit makes it easy to simulate multiple browser personalities, which is used in one of our detection features. For example, depending on the personality we want to assume, we can easily configure the value of HTTP headers that are transmitted with each request (e.g., the `User-Agent` header), the settings of JavaScript attributes (such as the `navigator` object and its properties), and the handling of certain HTML and JavaScript features and capabilities that are implemented differently in different browsers (e.g., event handlers are registered using the `addEventListener()` function in Firefox and the `attachEvent()` function in Internet Explorer). While some of these differences could be handled by existing browser's extensions (e.g., the User Agent plugin for Firefox [25]), others would require more substantial changes to the browser itself.

Second, in HtmlUnit, it is possible to simulate an arbitrary system environment and configuration. In fact, we have modified HtmlUnit so that, regardless of the actual system configuration, requests for loading any ActiveX control or plugin are successful and cause the instantiation of a custom logging object, which keeps track of all methods and attributes invoked or set on the control. This allows us to detect, without any further configuration effort, exploits that target any control or plugin, even those for which no vulnerability has been publicly disclosed. This is different from traditional high-interaction honeyclients (and real browsers), where

an actual component needs to be installed for the exploit to succeed and for the system to detect the attack.

A third reason for using HtmlUnit is that it allows us to implement anti-cloaking mechanisms. Malicious scripts sometimes employ techniques to masquerade their real behavior, for example, they launch an attack only if the language of a visitor's browser is `en-US`. In these cases, it would be beneficial to increment the code coverage of a script to potentially expose a more complete picture of its actions. Therefore, we implemented one such technique. More precisely, at run-time, we parse all the code provided to the JavaScript interpreter, and we keep track of all the functions that are defined therein. When the regular execution of the script finishes, we force the execution of those functions that have not been invoked, simply by calling them. While less sophisticated than other approaches with similar goals [19], this technique resembles the forced-execution model presented in [40]. We found that this simple technique worked well in practice.

4. ANALYSIS

We implemented our proposed approach in a system, which we call JSAND, and we used it to detect and analyze malicious web content. In this section, we describe some of the analyses that our system can perform on malicious JavaScript code.

Exploit classification. It is often useful to understand which vulnerabilities are exploited by a malicious page. We currently focus on vulnerabilities in browser plugins and ActiveX controls. JSAND extracts this information in two phases. The first phase consists of identifying exploits used in the wild. JSAND analyzes the samples that it flagged as malicious and collects all the events (method invocations and attribute settings) related to plugins and controls that were considered anomalous. For each event, JSAND extracts four *exploit features*: the name of the plugin, the name of the method or attribute involved, the position of the anomalous parameters (if any), and the type of the identified anomaly (e.g., long string value or anomalous character distribution). Note that by considering the anomaly type rather than the concrete, actual values used in an event, we abstract away from the concrete exploit instance. Then, for each feature set, we manually search vulnerability repositories, such as CVE, for vulnerability reports that match the set. If we find a match, we label the corresponding set with the identified vulnerability, and we say that the feature set characterizes an *exploit class*, i.e., the exploits for the matching vulnerability.

In the second phase, the actual classification is performed. This step is completely automatic. For each new sample that is analyzed, JSAND collects anomalous events related to plugins and extracts their exploit features. It then uses a naive Bayesian classifier to classify the exploit feature values in one of the exploit classes. If the classifier finds a classification with high confidence, the event is considered a manifestation of an exploit against the corresponding vulnerability. This classification is both precise and robust, since it is obtained from analyzing the behavior of a running exploit, rather than, for example, looking for a static textual match. Most tools we are aware of do not or cannot provide this kind of information to their users. The exceptions are anti-virus programs and PhoneyC, a low-interaction honeyclient (described later).

Signature generation. We can also use the exploit classification information to generate exploit signatures for signature-based tools. More precisely, we generate signatures for the PhoneyC tool. In PhoneyC, signatures are JavaScript objects that redefine the methods and attributes of a vulnerable component with functions that check if the conditions required for a successful exploit are met. In our experience, the information stored in our exploit classes is often sufficient to automatically generate high-quality signatures for PhoneyC. To demonstrate this, we generated signatures for three exploits that were not detected by PhoneyC and submitted them to the author of PhoneyC.

5. SYSTEM EVALUATION

We will now describe how JSAND performs at detecting pages that launch drive-by-download attacks. In particular, we examine the accuracy of our detection approach and compare it with state-of-the-art tools on over 140K URLs. Note, however, that we are not attempting to perform a measurement study on the prevalence of malicious JavaScript on the web (in fact, such studies have appeared before [21, 39], and have examined an amount of data that is not available to us [28]).

5.1 Detection Results

To evaluate our tool, we compiled the following seven datasets: a *known-good dataset*, four *known-bad datasets*, and two *uncategorized datasets*.

The *known-good dataset* consists of web pages that (with high confidence) do not contain attacks. We use this dataset to train our models, to determine our anomaly thresholds, and to compute false positives. In total, the dataset contains 11,215 URLs. We populated the known-good dataset by downloading the pages returned for the most popular queries in the past two years, as published by the Google and Yahoo! search engines, and by visiting the 100 most popular web sites, as determined by Alexa. This allowed us to obtain pages representative of today's use of JavaScript. Furthermore, we used the Google Safe Browsing API to discard known dangerous pages [13].

The *known-bad datasets* contain pages and scripts that are known to be malicious. We use these datasets to evaluate the detection capabilities of our tool and compute false negatives. In total, they consist of 823 malicious samples. These samples were organized in four different datasets, according to their sources:

- *The spam trap dataset.* From January to August 2008, we retrieved a feed of spam URLs provided by Spamcop [36]. For about two months, we also extracted the URLs contained in emails sent to a local spam trap. To distinguish URLs directing to drive-by-download sites from those URLs that simply lead to questionable sites (e.g., online pharmacies), we analyzed each URL with Capture-HPC, a high-interaction honeyclient system [37], which classified 257 pages as malicious. We manually verified that these pages actually launched drive-by downloads.
- *The SQL injection dataset.* From June to August 2008, we monitored several SQL injection campaigns against a number of web sites. These campaigns aimed at injecting in vulnerable web sites code that redirects the visitor's browser to a malicious page. We identified 351 domains involved in the attacks. From these, we collected 23 distinct samples.
- *The malware forum dataset.* We collected 202 malicious scripts that were published or discussed in several forums, such as `malwaredomainlist.com` and `milw0rm.com`.
- *The Wepawet-bad dataset.* This dataset contains the URLs that were submitted to our online service (`wepawet.cs.ucsb.edu`, Wepawet in short), which allows the public submission of URLs for analysis by JSAND. More precisely, we looked at 531 URLs that were submitted during the month of January 2009 and that, at the time of their submission, were found to be malicious by JSAND. We re-analyzed them to verify that the malicious code was still present and active on

those pages. We identified 341 pages that were still malicious.

The *uncategorized datasets* contain pages for which no ground truth is available. The uncategorized pages were organized in two datasets:

- *The crawling dataset*. This dataset contains pages collected during a crawling session. The crawling was seeded with the results produced by the Google, Yahoo!, and Live search engines for queries in a number of categories, which were also used in previous studies on malicious web content [14, 21, 28]. In total, we examined 115,706 URLs from 41,197 domains (to increase the variety of pages analyzed, we examined up to 3 pages per domain).
- *The Wepawet-uncat dataset*. This dataset contains 16,894 pages that were submitted to the Wepawet service between October and November 2009.

False positives. We randomly divided the known-good dataset in three subsets and used them to train JSAND and compute its false positive rate. More precisely, we ran JSAND on 5,138 pages to train the models. We then ran it on 2,569 pages to establish a threshold, which we set to 20% more than the maximum anomaly score determined on these pages. The remaining 3,508 pages were used to determine the false positive rate. JSAND caused no false positives on these pages.

In addition, we computed the false positive rate on the crawling dataset. This is a more extensive test both in terms of the number of pages examined (over 115K) and their types (e.g., these pages are not necessarily derived from popular sites or from results for popular search queries). On this dataset, JSAND reported 137 URLs as being malicious. Of these, we manually verified that 122 did actually launch drive-by downloads. The remaining 15 URLs (hosted on ten domains) appeared to be benign, and, thus, are false positives. The majority of them used up to 8 different ActiveX controls, some of which were not observed during training, yielding an anomaly score larger than our threshold.

False negatives. For the next experiment, we compared the detection capabilities of JSAND with respect to three other tools: ClamAV [2], PhoneyC [23], and Capture-HPC [37]. These tools are representative of different detection approaches: syntactic signatures, low-interaction honeyclients using application-level signatures, and high-interaction honeyclients, respectively.

ClamAV is an open-source anti-virus, which includes more than 3,200 signatures matching textual patterns commonly found in malicious web pages. We used ClamAV with the latest signature database available at the time of the experiments.

PhoneyC is a browser honeyclient that uses an emulated browser and application-level signatures. Signatures are expressed as JavaScript procedures that, at run-time, check the values provided as input to vulnerable components for conditions that indicate an attack. Thus, PhoneyC's signatures characterize the dynamic behavior of an exploit, rather than its syntactic features. In addition, PhoneyC scans pages with ClamAV. Unlike our tool, PhoneyC can only detect attacks for which it has a signature. We used PhoneyC version 1680, the latest available at the time of running the experiments.

Capture-HPC is a high-interaction honeyclient. It visits a web page with a real browser and records all the resulting modifications to the system environment (e.g., files created or deleted, processes launched). If any unexpected modification occurs, this is considered the manifestation of an attack launched by the page. We used the default configuration of Capture-HPC and installed Windows XP SP2 and Internet Explorer (a setup used in previous studies [24]). In addition, we installed the five plugins most targeted by exploits in the Wepawet-bad dataset, including vulnerable versions of Adobe Reader (9.0) and Flash (6.0.21).

Dataset	Samples (#)	JSAND FN	ClamAV FN	PhoneyC FN	Capture-HPC FN
Spam Trap	257	1 (0.3%)	243 (94.5%)	225 (87.5%)	0 (0.0%)
SQL Injection	23	0 (0.0%)	19 (82.6%)	17 (73.9%)	–
Malware Forum	202	1 (0.4%)	152 (75.2%)	85 (42.1%)	–
Wepawet-bad	341	0 (0.0%)	250 (73.3%)	248 (72.7%)	31 (9.1%)
Total	823	2 (0.2%)	664 (80.6%)	575 (69.9%)	31 (5.2%)

Table 2: Comparison of detection results on the known-bad datasets. FN indicates false negatives.

Table 2 shows the results of evaluating the detection effectiveness of the different approaches on the known-bad datasets. For these tests, we only report the false negatives (all samples are known to be malicious). We did not test Capture-HPC on the SQL injection dataset and the malware forum dataset, since the attacks contained therein have long been inactive (e.g., the web sites that hosted the binaries downloaded by the exploit were unreachable). Thus, Capture-HPC would have reported no detections.

JSAND had two false negatives on the known-bad datasets. This corresponds to a false negative rate of 0.2%. The undetected exploits do not use obfuscation and attack a single vulnerability in one ActiveX control. JSAND detected anomalies in the number of memory allocations (due to heap spraying) and in the instantiation of a control that was not observed during training, but this was not sufficient to exceed the threshold.

ClamAV missed most of the attacks (80.6%). While better results may be obtained by tools with larger signature bases, we feel it is indicative of the limitations of approaches based on static signature matching. The most effective signatures in ClamAV matched parts of the decoding routines, methods used in common exploits, and code used to perform heap spraying. However, these signatures would be easily evadable, for example, by introducing simple syntactic modifications to the code.

PhoneyC had almost a 70% false negative rate, even if it has signatures for most of the exploits contained in the known-bad datasets. At first inspection, the main problem seems to be that PhoneyC (in the version we tested) only handles a subset of the mechanisms available to execute dynamically generated JavaScript code. If unsupported methods are used (e.g., creating new `script` elements via the `document.createElement` method), PhoneyC does not detect the attack.

Capture-HPC missed 9.1% of the attacks in the Wepawet-bad dataset (and 5.2% overall). We manually analyzed the URLs that were not detected as malicious and we found that, in most cases, they were using exploits targeting plugins that were not installed on our Capture-HPC system. This result highlights that the configuration of the environment used by Capture-HPC is a critical factor in determining its detection rate. Unfortunately, installing all possible, vulnerable components can be difficult, since targeted vulnerabilities are scattered in tens of different applications, which must be correctly installed and configured (for example, the known-bad datasets include 51 different exploits, targeting vulnerabilities in 40 different components).

To better understand this issue, we computed how the detection rate of Capture-HPC would change on the Wepawet-bad dataset, given different sets of ActiveX controls. In particular, we want to understand what is the minimum number of components that needs to be installed to achieve a given detection rate. This question can be cast in terms of the set cover problem, where installing an application guarantees the detection of all pages that contain at least one

exploit targeting that application. In this case, we say that the application "covers" those pages. "Uncovered" pages (those that do not target any of the installed applications) will not be detected as malicious. To solve the problem, we used the greedy algorithm (the problem is known to be NP-complete), where, at each step, we add to the set of installed applications a program that covers the largest number of uncovered pages. Figure 1 shows the results. It is interesting to observe that even though a relatively high detection rate can be achieved with a small number of applications (about 90% with the top 5 applications), the detection curve is characterized by a long tail (one would have to install 22 applications to achieve 98% of detection). Clearly, a false negative rate between 10% and 2% is significant, especially when analyzing large datasets.

Large-scale comparison with high-interaction honeyclients. We performed an additional, more comprehensive experiment to compare the detection capability of our tool with high-interaction honeyclients. More precisely, we ran JSAND and Capture-HPC side-by-side on the 16,894 URLs of the Wepawet-uncat dataset. Each URL was analyzed as soon as it was submitted to the Wepawet online service. Capture-HPC and JSAND were run from distinct subnets to avoid spurious results due to IP cloaking.

Overall, Capture-HPC raised an alert on 285 URLs, which were confirmed to be malicious by manual analysis. Of these, JSAND missed 25. We identified the following reasons for JSAND's false negatives. In four cases, JSAND was redirected to a benign page (`google.cn`) or an empty page, instead of being presented with the malicious code. This may be the result of a successful detection of our tool or of its IP. An internal bug caused the analysis to fail in three additional cases. Finally, the remaining 18 missed detections were the consequence of subtle differences in the handling of certain JavaScript features between Internet Explorer and our custom browser (e.g., indirect calls to `eval` referencing the local scope of the current function) or of unimplemented features (e.g., the `document.lastModified` property).

Conversely, JSAND flagged 8,714 URLs as anomalous (for 762 of these URLs, it was also able to identify one or more exploits). Of these, Capture-HPC missed 8,454. We randomly sampled 100 URLs from this set of URLs and manually analyzed them to identify the reasons for the different results between JSAND and Capture-HPC. We identified three common cases. First, an attack is launched but it is not successful. For example, we found many pages (3,006 in the full Wepawet-uncat dataset) that were infected with JavaScript code used in a specific drive-by campaign. In the last step of the attack, the code redirected to a page on a malicious web site, but this page failed to load because of a timeout. Nonetheless, JSAND flagged the infected pages as anomalous because of the obfuscation and redirection features, while Capture-HPC did not observe the full attack and, thus, considered them benign. We believe JSAND's behavior to be correct in this case, as the infected pages are indeed malicious (the failure that prevents the successful attack may be only temporary). Second, we noticed that Capture-HPC stalled during the analysis (there were 1,093 such cases in total). We discovered that, in some cases, this may be the consequence of an unreliable exploit, e.g., one that uses too much memory, causing the analyzer to fail. Finally, a missed detection may be caused by evasion attempts. Some malicious scripts, for example, launch the attack only after a number of seconds have passed (via the `window.setTimeout` method) or only if the user minimally interacts with the page (e.g., by releasing a mouse button, as it is done in the code used by the Mebroot malware). In these cases, JSAND was able to expose the complete behavior of the page thanks to the forced execution technique described in Section 3.

Performance. JSAND's performance clearly depends on the complexity of the sample under analysis (e.g., number of redirects and scripts to interpret). However, to give a feeling for the overall performance of our tool, we report here the time required to analyze the Wepawet-bad dataset. JSAND completed the workload in 2:22 hours. As a comparison, Capture-HPC examined the same set of pages in 2:59 hours (25% slower). The analysis can be easily parallelized, which further improves performance. For example, by splitting the workload on three machines, JSAND completed the task in 1:00 hour. While our browser and JavaScript interpreter are generally slower than their native counterparts, we have no overhead associated with reverting the machine to a clean state after a successful exploitation, as required in Capture-HPC.

Features and Anomaly Score. We analyzed the impact that each feature group has on the final anomaly score of a page. We found that, on the known-bad datasets, exploitation features account for the largest share of the score (88% of the final value), followed by the environment preparation features (9%), the deobfuscation features (2.7%), and the redirection and cloaking features (0.3%). The contribution of redirection and cloaking features is limited also because the majority of the samples included in the known-bad datasets consist of self-contained files that do not reference external resources and, therefore, cause no network activity.

Furthermore, we examined the breakdown of the anomaly score between necessary and useful features. Figure 2 shows the results. In the figure, samples are ordered according to their overall anomaly score. The contribution of useful features to the final value is plotted against the left axis and the contribution of necessary features is plotted against the right axis. It can be seen that the necessary features clearly dominate the final anomaly score (note that the necessary feature axis is log scale). In particular, the use of multiple exploits in the same page causes the anomaly score to "explode." For example, the total anomaly score of 103 samples was more than ten times higher than the threshold. This is good because it demonstrates that the detection of malicious pages heavily relies on those features that are fundamental to attacks. Nonetheless, useful features contribute positively to the final score, especially for samples where the anomaly value determined by necessary features is low (this is the case for scores represented in the lower left corner of the graph). In particular, without the contribution of useful features, the anomaly score of seven samples would be below the threshold (i.e., they would be false negatives).

Summary. In summary, our results indicate that JSAND achieves a detection rate that is significantly better than state-of-the-art tools. It detected a large number of malicious pages not detected by other tools, and it missed only a very limited number of attacks. It also raised only a few false positives. Furthermore, in our tests, JSAND analyzed about two samples per minute, which is faster than other tools. Since the analysis can be easily parallelized, performance can be further improved.

5.2 Operational Experience

We made JSAND publicly available at `http://wepawet.cs.ucsb.edu` as an online service, where users can submit URLs or files for analysis. For each sample, a report is generated that shows the classification of the page and the results of the deobfuscation, exploit classification, and other analyses.

The service has been operative since November 2008, and it is used by a growing number of users. For example, in October 2009, it was visited by 10,598 unique visitors (according to Google Analytics data), who submitted 37,547 samples for analysis. Of these users, 97 were frequent users, i.e., submitted more than 30 samples. Of all samples analyzed in October, 11,679 (31%) were flagged as

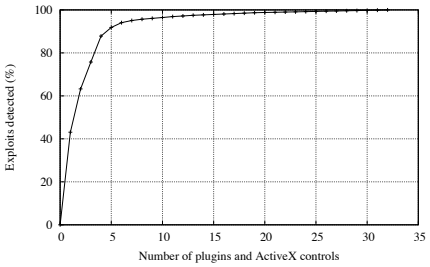

Figure 1: Capture-HPC detection rate as a function of the installed vulnerable applications.

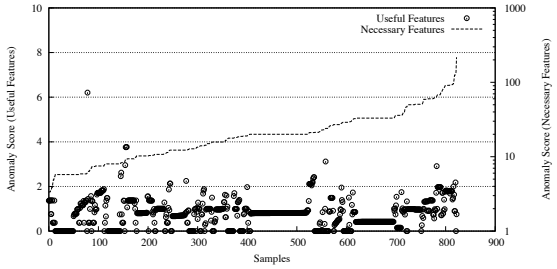

Figure 2: Breakdown of the anomaly score for samples in the known-bad datasets by feature category.

malicious (after additional analysis, we attributed 4,951 samples to one of eleven well-known drive-by campaigns). Our tool was able to classify the exploits used in 1,545 of the malicious samples.

The reports generated by JSAND are routinely used to investigate incidents and new exploits (e.g., [1]), as supportive evidence in take-down requests, and to complement existing blacklists (e.g., in October 2009 alone, it detected drive-by downloads on 409 domains that at the moment of the analysis were not flagged as malicious by Google Safe Browsing [13]).

6. POSSIBLE EVASION

It is interesting to discuss more in detail how JSAND performs in terms of resilience to evasion attempts.

Novel attacks. One way for attackers to evade our detection is to launch attacks that completely bypass the features we use. We have shown that our features characterize very well the types of attacks that are performed today. Thus, attackers would need to find completely new attack classes. In particular, we note that JSAND is capable of detecting previously-unseen attacks, when these attacks manifest in anomalous features. For example, JSAND correctly flagged as malicious scripts used in the recent "Aurora" attack [12]. Even if the exploits target a vulnerability (CVE-2010-0249) in a component of Internet Explorer (the memory manager) that is not characterized by a specific feature, the anomaly scores associated with the shellcode detection and environment preparation were sufficient to raise an alert. In general, to detect a 0-day attack, other tools would need to upgrade their signature database or install the additional vulnerable component(s).

Emulation fingerprinting. A second possible evasion technique consists of detecting differences between JSAND's emulated environment and a real browser's environment. Malicious scripts may fingerprint the browser (e.g., they can check the headers our tool sends and the JavaScript objects it provides), its plugins and ActiveX controls (e.g., to test that they actually expose the expected methods and parameters), or the JavaScript interpreter to identify any differences. This is an issue that is common to any emulated environment and that affects, to a certain degree, even high-interaction honeyclients using virtual machines [3].

We have two ways of counteracting this technique. First, we can set up our environment so that it behaves as accurately as possible as the browser we want to impersonate. This clearly becomes an arms race between the attacker's fingerprinting efforts and our emulation efforts. However, the underlying browser we use, HtmlUnit, is designed to take into account the different behaviors of different browsers and, in our experience, it was generally easy to correct the deviations that we found.

The second technique to counteract evasion consists of forcing the execution of a larger portion of a script to uncover parts of the code (and, thus, behaviors) that would otherwise be hidden. These correspond, for example, to functions containing exploit code that are not invoked unless the corresponding vulnerable component is identified in the browser. We have already discussed our simple method to increase the executed code coverage.

7. RELATED WORK

A number of approaches and tools have been proposed in recent years to identify and analyze malicious code on the web. We will now briefly present the most relevant ones and compare them with our approach.

System state change. A number of approaches (e.g., HoneyMonkey [39], Capture-HPC [37], Moshchuk et al. [20, 21], and Provos et al. [28]) use high-interaction honeyclients to visit potentially-malicious web sites and monitor changes in the underlying operating system that may be caused by malicious web pages. The system change approach gives detailed information about the consequences of a successful exploit, e.g., which files are created or which new processes are launched. However, it gives little insight into how the attack works. In addition, it fails to detect malicious web content when the honeyclient is not vulnerable to the exploits used by the malicious page (e.g., the vulnerable plugins are not installed). As we have seen, JSAND solves this problem by simulating the presence of any ActiveX control or plugin requested by a page.

Generic malware tools. Some tools (e.g., Monkey-Spider [14] and SpyeBye [27]) consider a page to be malicious if it links to resources that are classified as malware by external tools, such as anti-virus, malware analysis tools, or domain blacklists. JSAND can be used in these approaches as an additional detector.

Malicious code signatures. Signatures (i.e., patterns that characterize malicious code) can be matched at the network level (e.g., in the Snort IDS [31]), or at the application level (as in PhoneyC [23]). JSAND does not rely on predetermined signatures, and, thus, can potentially detect novel attacks. Furthermore, we have shown that JSAND can automatically generate signatures for signature-based systems.

Anomaly detection. Caffeine Monkey is a tool to collect events associated with the execution of JavaScript code [7]. The authors propose to use the distribution of function calls in a JavaScript program to differentiate between malicious and benign pages. However, their detection technique is based on the manual analysis of the collected events and their validation only considered 368 scripts. JSAND has more comprehensive features, is automatic, and has been extensively validated.

Shellcode detection. A number of recent approaches are based on the detection of the shellcode used in an attack (e.g., [6, 30]). Two of JSAND's features are designed for the same purpose. Furthermore, the richer feature set of JSAND allows it to detect attacks that

do not rely on shellcode injection, which would necessarily evade detection by these approaches.

8. CONCLUSIONS

We presented a novel approach to the detection and analysis of malicious JavaScript code that combines anomaly detection techniques and dynamic emulation. The approach has been implemented in a tool, called JSAND, which was evaluated on a large corpus of real-world JavaScript code and made publicly available online. The results of the evaluation show that it is possible to reliably detect malicious code by using emulation to exercise the (possibly hidden) behavior of the code and comparing this behavior with a (learned) model of normal JavaScript code execution.

Future work will extend the techniques described here to improve the detection of malicious JavaScript code. For example, we plan to improve the procedures to identify binary shellcode used in JavaScript malware. In addition, we plan to implement a browser extension that is able to use the characterization learned by JSAND to proactively block drive-by-download attacks.

Acknowledgment

This work has been supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, CCR-0716095, CCR-0831408, CNS-0845559 and CNS-0905537, and by the ONR under grant N000140911042. We would like to thank Jose Nazario and the malwaredomainlist.com community for providing data.

9. REFERENCES

[1] Andre L. IE 0day exploit domains. http://isc.sans.org/diary.html?storyid=6739, 2009.

[2] ClamAV. Clam AntiVirus. http://www.clamav.net/.

[3] D. De Beer. Detecting VMware with JavaScript. http://carnal0wnage.blogspot.com/2009/04/, 2009.

[4] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2008.

[5] D. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[6] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2009.

[7] B. Feinstein and D. Peck. Caffeine Monkey. http://www.secureworks.com/research/tools/caffeinemonkey.html.

[8] S. Frei, T. Dübendorfer, G. Ollman, and M. May. Understanding the Web browser threat: Examination of vulnerable online Web browser populations and the "insecurity iceberg". In *Proceedings of DefCon 16*, 2008.

[9] Gargoyle Software Inc. HtmlUnit. http://htmlunit.sourceforge.net/.

[10] D. Goodin. Department of Homeland Security website hacked! http://www.theregister.co.uk/2008/04/25/mass_web_attack_grows/, 2008.

[11] D. Goodin. SQL injection taints BusinessWeek.com. http://www.theregister.co.uk/2008/09/16/businessweek_hacked/, 2008.

[12] D. Goodin. Exploit code for potent IE zero-day bug goes wild. http://www.theregister.co.uk/2010/01/15/ie_zero_day_exploit_goes_wild/print.html, 2010.

[13] Google. Safe Browsing API. http://code.google.com/apis/safebrowsing/.

[14] A. Ikinci, T. Holz, and F. Freiling. Monkey-Spider: Detecting Malicious Websites with Low-Interaction Honeyclients. In *Proceedings of Sicherheit, Schutz und Zuverlässigkeit*, April 2008.

[15] Internet Security Systems X-Force. Mid-Year Trend Statistics. Technical report, IBM, 2008.

[16] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2003.

[17] C. Kruegel, G. Vigna, and W. Robertson. A Multi-model Approach to the Detection of Web-based Attacks. *Journal of Computer Networks*, 48(5), July 2005.

[18] MITRE Corporation. Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org/.

[19] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[20] A. Moshchuk, T. Bragin, D. Deville, S. Gribble, and H. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. In *Proceedings of the USENIX Security Symposium*, 2007.

[21] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware in the Web. In *Proceedings of the Symposium on Network and Distributed System Security*, 2006.

[22] Mozilla.org. Rhino: JavaScript for Java. http://www.mozilla.org/rhino/.

[23] J. Nazario. PhoneyC: A Virtual Client Honeypot. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

[24] New Zealand Honeynet Project. Know Your Enemy: Malicious Web Servers. http://www.honeynet.org/papers/mws, 2007.

[25] C. Pederick. User Agent Switcher Firefox Plugin. https://addons.mozilla.org/en-US/firefox/addon/59.

[26] M. Polychronakis, P. Mavrommatis, and N. Provos. Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2008.

[27] N. Provos. SpyBye. http://code.google.com/p/spybye.

[28] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the USENIX Security Symposium*, 2008.

[29] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnet*, 2007.

[30] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the USENIX Security Symposium*, 2009.

[31] M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Conference on System Administration*, 1999.

[32] C. Seifert, I. Welch, P. Komisarczuk, C. Aval, and B. Endicott-Popovsky. Identification of Malicious Web Pages Through Analysis of Underlying DNS and Web Server Relationships. In *Proceedings of the Australasian Telecommunication Networks and Applications Conference*, 2008.

[33] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. http://www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php, 2004.

[34] A. Sotirov. Heap Feng Shui in JavaScript. Black Hat Europe, 2007.

[35] A. Sotirov and M. Dowd. Bypassing Browser Memory Protections: Setting back browser security by 10 years. Black Hat, 2008.

[36] SpamCop. SpamCop.net. http://www.spamcop.net/, 2008.

[37] The Honeynet Project. Capture-HPC. https://projects.honeynet.org/capture-hpc.

[38] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2002.

[39] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security*, 2006.

[40] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection*, 2007.