ADSandbox: Sandboxing JavaScript to fight Malicious Websites

Andreas Dewald
Laboratory for Dependable Distributed Systems
University of Mannheim
dewald@informatik.uni-mannheim.de

Thorsten Holz*
International Secure Systems Lab
Technical University of Vienna
tho@iseclab.org

Felix C. Freiling
Laboratory for Dependable Distributed Systems
University of Mannheim
freiling@informatik.uni-mannheim.de

ABSTRACT

We present ADSandbox, an analysis system for malicious websites that focusses on detecting attacks through JavaScript. Since, in contrast to Java, JavaScript does not have any built-in sandbox concept, the idea is to execute any embedded JavaScript within an isolated environment and log every critical action. Using heuristics on these logs, ADSandbox decides whether the site is malicious or not. In contrast to previous work, this approach combines generality with usability, since the system is executed directly on the client running the web browser before the web page is displayed. We show that we can achieve false positive rates close to 0% and false negative rates below 15% with a performance overhead of only a few seconds, what is a bit high for real time application, but supposes a great potential for future versions of our tool.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and Protection*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive Software*

General Terms

Security

Keywords

Malicious software, Drive-by downloads, Dynamic analysis

1. INTRODUCTION

The increasing danger of malicious websites.

Today it is much more difficult for classical malicious software (malware) like worms, bots or trojans to spread within the network using common vulnerabilities of operating systems. This is partly caused by technical progress in operating systems engineering as well as an increasing awareness of Internet users to keep their operating systems up to date. This has shifted the focus of attackers to application level vulnerabilities that currently are still more easily exploitable. From the possible set of vulnerable applications, web browsers are currently the most popular targets. Today, attackers create "malicious" websites that try to exploit vulnerabilities in the visitor's web browser. These malicious sites try to distribute malware and through such *drive-by downloads* gain control over the machine.

Much of the dangers of malicious websites stem from the ubiquitous usage of JavaScript. Although the execution of JavaScript can be turned off in every web browser, almost all (popular) websites today contain JavaScript and so many users have it enabled by default. This is problematic, since JavaScript opens up several possibilities of stealing information and exploiting browser vulnerabilities to an attacker. For example, an attacker could read the victims cookies and send it to a drop-zone through a simple http redirect. A major threat are heap spraying attacks, which exploit memory corruption vulnerabilities of the browser or installed browser-plugins by allocating big amount of memory on the heap, usually utilising arrays of strings that each contain shellcode and a prepended NOP-slide.

Previous work.

Most approaches to counter malicious websites today are *infrastructure-based* [7, 10, 11]. These systems continuously crawl the web and analyse the visited websites using different heuristics. The outcome of this analysis (malicious or not) is stored in a central database. Before a user visits a website, the web browser (through a special plug-in) queries the database for the classification of the target URL. If it has been classified as malicious, the user is redirected to a website that shows an according warning.

The main advantage of this approach is that it is instantly usable by end-users. The approach is also rather flexible since it does not restrict the ways in which malicious websites are analyzed and detected, i.e., it is able to cover a broad range of malicious behavior. However, the approach relies on a dedicated infrastructure, i.e., crawlers to analyse websites and a central database to which web browsers can

*Thorsten Holz is also affiliated with the Laboratory for Dependable Distributed Systems at University of Mannheim.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

connect. It also suffers from inconsistency issues because the user is warned based on the classification that is stored in the central database. There is no guarantee that the website visited by the user has not been compromised since the last analysis. Another problem of this approach are exploits that are triggered not on every visit.

As mentioned above, the success of the infrastructure-based approach depends heavily on the quality of website analysis after the site has been crawled. Wang et al. [16, 15] present a system utilising a *honeypot* within a virtual machine (*VM*). Within the VM, the tool creates an instance of the Microsoft Internet Explorer, navigates to a specific URL, and waits for a few minutes. Any changes to the VM's file system and registry are flagged. If a change to the file system outside of the temporary directory of the browser is detected, the site is classified as malicious. The system then shuts down the potentially infected VM and starts a clean one to analyse the next URL. To determine if an exploit just works against a specific combination of versions of Microsoft Internet Explorer and the underlying operating system (or if it even is a zero-day exploit), a pipeline of such virtual machines is used, each covering a different patch level.

A general problem of this approach is that it does not scale well. Although the analysis effort can be nicely parallelized on redundant hardware, analysing millions of websites per day can be rather tedious and costly. The analysis frequency however is vital to the degree of protection of the system: If a website becomes malicious, it should be crawled and analyzed as quickly as possible. Even if every site on the Internet is visited once per day (an almost impossible task), this still leaves windows of vulnerability in which users are not protected. This problem is amplified by the fact that malicious websites do not necessarily trigger an exploit every time they are visited. In addition, exploits may trigger on user interaction only.

Cox et al. [4] proposed to move the VM from the infrastructure to the user, thus partly leveraging the problem of scalability. This introduces *client-side approaches* to counter malicious websites. However, end users now need to invest in (hardware) resources to run one virtual machine (with the patch level used on the productive system that has to be protected) and to set up the honeypot and additional control software. The overhead due to Cox et al. is 1.06 seconds if there already is a pre-forked VM with the corresponding guest operating systems and browser. In contrast, the overhead to start a new VM is 9.26s. So even the approach by Cox et al. [4] suffers from rather high overhead and low usability.

Since they were primarily intended to find zero day exploits, the honeypot approaches described above [16, 15, 4] can also be restricted in their generality. Two recent papers [13, 6] present systems that specialise on detecting heap spraying attacks. The idea is to intercept calls to the Mozilla Firefox memory manager and to observe objects on the heap. The Nozzle system [13] for example searches for valid x86 code on the heap using a lightweight emulator. Valid code sequences are disassembled and a control flow graph is built, which can then be analyzed using methods known from sled detection in network packet processing [1, 9]. Unfortunately, this approach alone has a high false positive rate, because many objects look like valid x86 code as a result of the density of the x86 instruction set. This is why the detection is usually accompanied by a global "heap health metric" [13].

Sandboxing JavaScript.

Ideally, client-side detection techniques should be integrated into a browser and thus should be *transparent* to the user. However, they should also be sufficiently *general*, i.e., able to detect a wide range of attacks, not only those that spray the heap. In this paper we present ADSandbox, a client-side detection approach that is both transparent and general, in the sense that it allows to detect a large class of JavaScript-based attacks (not only heap spraying). The idea is not to run the entire website (including the browser) in a controlled environment but only the JavaScript programs.

The core of ADSandbox is a controlled execution environment (sandbox) for JavaScript. It utilises the Mozilla JavaScript engine *SpiderMonkey* to execute JavaScript programs and log every action during the execution. Afterwards, the system uses heuristics on the resulting log to detect malicious behaviour. All this is implemented within a *browser helper object* (*BHO*) for the Microsoft Internet Explorer. This BHO interrupts any navigation process and initiates the analysis of the target URL by the sandbox. Whenever malicious content is detected, the user is warned by the BHO and the malicious content is blocked unless the user explicitly wants to load it anyway. To support the user in this decision a short report, naming the detected threats, is displayed.

The sandbox and the analysis framework are implemented as a standalone dynamic link library (DLL) to ease the development of similar plug-ins for other browsers. To detect a broad range of attacks, we implemented different analysis levels. At the lowest level, we perform a static analysis of the source code, searching (for instance) for hidden IFrames. The next levels involve dynamic analysis of the behavior of the JavaScript programs embedded in the website. On a reasonable set of sample sites the system had a false positive rate of 0% and a false negative rate between 6 and 13%. In most cases websites are analyzed rather quickly: 90% of the chosen websites were analyzed in less than 5 seconds on a standard desktop PC. We consider this to be quite acceptable for a first, unoptimized prototype implementation. We therefore believe that with some additional performance-tuning effort our approach has the potential to be of use in tools that analyse websites on the fly when browsing.

Contributions.

To summarise, this paper presents the following contributions:

- We present ADSandbox, an analysis system for malicious websites before the website is displayed in the browser.
- In contrast to previous work, this approach is much more general. It can detect many different forms of malicious JavaScript (for example heap spraying attacks and others).
- In contrast to previous work, our system combines generality with usability since it is executed directly in real time on the client running the web browser.

Paper outline.

The paper is structured as follows: In Section 2 we introduce the design of ADSandbox and explain the static and

dynamic analysis performed by the system in detail. We present in Section 3 the evaluation results, discuss the performance overhead of the system, and display noteworthy results obtained when studying thousands of live websites. Finally, we conclude the paper in Section 4. More information about ADSandbox can also be found elsewhere [5].

2. DESIGN OF THE SYSTEM

We now present an overview over the system together with some details about the design of the analysis systems.

2.1 System Overview

Our system is structured into three main components (see Fig. 1):

1. the browser helper object,
2. the wrapper executable, and
3. the actual analysis engine in a separate dynamic link library.

We explain the role of these components using the perspective of the user. The user can interact with the system in different ways. The most common way will be to access the system transparently through the BHO while surfing the web. This is depicted in the right hand side of Fig. 1. The sequence of actions and the control flow is denoted using numbers in dotted circles. First, the user makes use of his browser to visit a website. Then, before navigating to this website, the browser invokes the BHO. The BHO hands over the URL of the website to the analysis DLL. This DLL accesses the Internet to download the source of the desired website in order to analyse it. After the analysis has finished, the resulting report is returned to the BHO, which in turn displays this report within the browser if the website has been suspected. If the website has not been suspected or the user wants to visit it anyway, the browser resumes the navigation process to the target website. If, on the other hand, the website should not be visited due to a suspicion, the browser is navigated to a custom error page by the BHO.

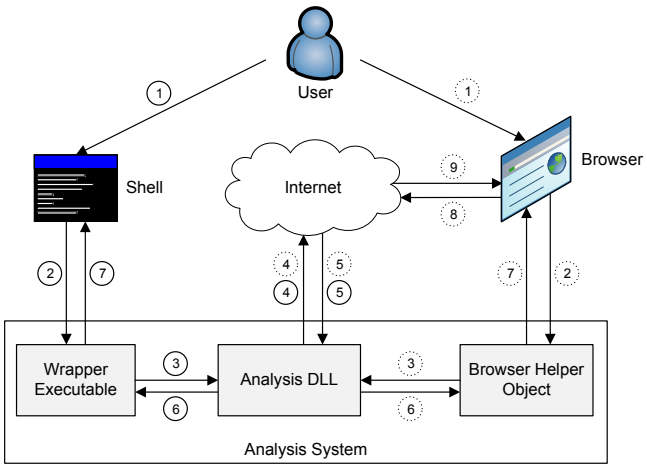

Figure 1: Schematic system overview of ADSandbox.

A second way to access the system is to use a wrapper executable in which a specific URL can be manually analysed in detail. The user starts the wrapper executable program on a command line, supplying some parameters to indicate which URL should be analysed and what kind of analysis should be performed. As shown on the left hand side of Fig. 1 (using numbers in solid circles), the wrapper executable then hands over the URL to the analysis DLL just like in the case of the BHO. After downloading the source code of the website and analysing it, the DLL's main analysis function again returns the report, which then is printed on the command line.

The wrapper executable offers slightly more options for the user to analyse websites: for example, a batch processing feature is able to process a list of URLs and to create detailed statistics about the analysed sites (see Section 3). The system can also be configured in such a way that even more detailed information about the analysis, such as an execution log of the dynamic JavaScript analysis, is delivered. This allows different applications of the analysis system using the wrapper executable, for example by integrating it into other toolsets.

2.2 Static Analysis

We implemented two static analysis methods, that each analyse the source code of the given website: The *static IFrame/same origin analysis* and the *static JavaScript analysis*. The first one detects every IFrame on a website and analyses several properties to decide whether this IFrame comes from a website with malicious intent. For example, settings that make the IFrame invisible to users, like a height or width of zero or a position that is out of the visible area, are facts that suggest maliciousness. Relevant is also the target URL of such an IFrame, which is another important indication in this context. There are four possible outcomes of this check:

1. The IFrame is hidden and refers to another domain,
2. it is hidden but loads an address in the same domain,
3. it is not hidden but refers to a foreign domain, or
4. it is neither hidden nor does it refer to another domain.

The only case we consider really suspicious is the one in which the IFrame is hidden and it refers to a foreign domain. What we try to detect is the unnoticed download of contents from another domain while visiting a website we trust. This aims on compromised trusted websites that are used by attackers to spread their malware through hidden IFrames. Thus we consider a hidden IFrame as not suspicious unless it refers to a foreign domain. In the case the IFrame is visible but refers to another domain, we are not able to decide if this happens with malicious intent or not. Thus we rely on the fact that the content of this IFrame is analysed itself as soon as it is loaded by Internet Explorer, as well.

2.3 Dynamic JavaScript Analysis

The static JavaScript analysis tries to detect malicious code via static analysis. We implemented several heuristics that were able to detect the manipulation of an object's prototype or the use of the `eval` function. But it soon became clear that these heuristics match far too often, because in contrast to our expectations many non-malicious websites also show such behavior as we discuss later in Section 3.

Thus, we feel that *dynamic* analysis is much more promising, as we now explain.

The dynamic JavaScript analysis is the core of our system to detect malicious websites. The main advantage of dynamic analysis is that we are able to analyse obfuscated JavaScript, too. This is very important, since most JavaScript based exploits currently observed in the wild try to hide their presence using several obfuscation techniques [2, 13, 14]. Usually obfuscation in JavaScript is reached through escaping or encoding the actual script. This code is then unescaped or decoded and executed by the JavaScript `eval` function. This procedure is often done several times recursively and thus it is quite some work to understand what the JavaScript actually does. But it is usually even impossible to automatically analyse such a JavaScript, because of the variety of available obfuscation methods. We show in this paper that this is possible with our dynamic JavaScript analysis, as each level of unpacking the obfuscated code is done just as it would be done in the browser of an attacked user. Additionally, it ought to be easier to detect malicious JavaScript based on its behavior than on its source code.

The analysis is performed in several phases: first, we extract the JavaScript source code from the HTML code and then create a *JavaScriptExecution* object, to which we pass the JavaScript source code. The JavaScriptExecution creates an instance of SpiderMonkey and executes the given JavaScript. SpiderMonkey on the other hand is modified each time an object is accessed to call a corresponding static callback function of the JavaScriptExecution class. With the help of this instrumentation every access to every JavaScript object is recognised and logged.

For our analysis, it is very important to know where the objects that are accessed originally come from. To supply this information we taint the JavaScript objects to also study how the objects are processed during the execution. For this kind of taint tracking [8] we make use of the *private data* pointer that every JavaScript object, whose class has the `JSCLASS_HAS_PRIVATE` flag set, has in SpiderMonkey. This pointer is especially meant to be used by C/C++ programs to store data for an object or respectively associate stored data with an object [12]. As this data is a C `void` pointer and is not visible or even accessible to the JavaScript program, it supplies an optimal way to taint each object with the name under which it has been accessed for the first time or even before it is accessed with a fixed name. Some objects may not have the private flag set. To keep track of such custom objects that are created within a JavaScript execution, we found an alternative way of object "tainting": the idea is to store the pointer of such an object in a map together with its name. Obviously this is not very reliable, because the pointer can change during a JavaScript execution, e.g., when the object is copied. But our experience shows that this way of taint tracking is often sufficient in practice.

Examples.

After the execution of the JavaScript we end up with the access log, which details all the behavior we observed during the execution of the JavaScript code. To illustrate how these logs look like we run the following JavaScript program within our sandbox:

```
var mystring = "sometext";
alert(mystring);
```

The resulting log shows how access to any object is logged:

```
ADD global.mystring
SET global.mystring TO "sometext"
GET global.alert
GET global.mystring
CONVERT alert TO A FUNCTION
FUNCTIONCALL alert ("sometext")
```

We distinguish between `ADD`, `DELETE`, `GET` and `SET` operations. Additionally we log any data type conversion, as well as function calls.

This execution log is then searched for patterns that reveal typical malicious behaviour. These patterns are implemented as regular expressions, which allows efficient matching using the standard PCRE implementation. For example, we search in the execution log if the JavaScript uses the function `SaveToFile` or `run`, which are often used combined, to save bytecode into a file and then run this file. The corresponding pattern is `CONVERT ([^\n]+\\.|)(SaveToFile|Run) TO A FUNCTION`.

In total, we implemented seven patterns to detect a broader range of attacks such as cookie stealing, file downloads and heap-spraying attacks, for example. Our evaluation results in the next section show that this covers typical malicious behavior found in the wild. Furthermore, patterns can be adapted and extended quite easily.

3. EVALUATION RESULTS

In this section, we present some results from the evaluation of our system on real-world conditions.

3.1 False Positive Rate

To measure the performance and the false positives rate of our system, we analysed the landing page of the top 1,000 websites from Alexa.com, which we suppose to be a good indication for the detection effectiveness of our tool. As this list is updated daily, we refer to the list of May 17, 2009 for our benchmark. Our system achieves a false positives rate of 0%, as none of these websites was suspected by our analysis.

3.2 False Negative Rate

We analysed 140 samples of potentially malicious websites from the wild. To gather these, we deployed several high interaction client honeypots [3], which surf the web and collect samples of every website that exhibits malicious behaviour. In addition, we manually extracted malicious code from well known malware construction kits.

Our system flagged 78 samples to be malicious websites, of which 15 were heap spraying attacks as shown in Table 1. The other 63 samples mostly contained JavaScript that uses `document.write` to write a hidden IFrame to include an exploit. But we also saw a couple of other exploits, too. For example, one of the samples tried to add an image and set its `src` property to a local DLL file on the user's hard disk, to load this DLL and exploit a vulnerability in it. Other 31 samples contained JavaScript that just redirected to another URL, where probably the actual exploit was hosted originally. Due to the fact that actually there was no exploit that could have been detected in these samples, we consider these not to be false negatives.

Unfortunately, we found that 10 samples could not be successfully analysed due to JavaScript errors, which we denote as potential false negatives. Another 21 samples that

Total			**140**
of which	Malicious		78
	of which	Heap Spraying	15
	of which	Other	63
of which	Redirects		31
of which	Not suspected		21
	of which	*Runs Plugin Test*	*5*
	of which	*DOM-Obfuscation*	*4*
	of which	Exploit Broken	4
	of which	No Exploit at all	8
of which	*JS Errors*		*10*

Table 1: Suspicion overview. Items in italics are considered false negatives.

we were able to analyse were not classified as malicious by our analysis system. Five of these samples could not be detected to be malicious because they check whether specific browser plug-ins are available before they trigger an exploit. As we currently do not emulate such browser behaviour, the exploit considers the plug-ins not to be available and thus does not trigger. Furthermore, there were four samples that used the DOM tree for obfuscation: we are for now not able to analyse this kind of obfuscated exploits due to the missing DOM tree in our analysis environment. Thus we have to denote another 9 false negatives from these two problems that are both caused by the lack of a complete browser emulation environment. An additional four samples did not contain a complete exploit and did just construct shellcode without using it. The other eight samples did not even contain an exploit. Finally, we end up with a false negative rate between 6,43 and 13,57% depending on if we consider the JavaScript errors as false negatives or not. Although we are already able to detect a lot of exploits, this shows that there is still some potential left for improvement.

3.3 Computational Overhead

Another important metric is the overhead of our analysis and therefore we also recorded the analysis processing time of each of the Alexa.com top 1,000 websites. The platform for our benchmark was an Intel Core2 Duo (6300 @ 1.86GHz) System with 2GB of RAM and Windows XP Professional SP2 installed. We measured the analysis time only without the time it takes to download the content of the sites, since this is a more representative analysis: the time needed for downloading content exhibits quite strong variation due to bandwidth and latency differences, depending on the Internet connection and routing paths. We measured an average processing time of 2,112 milliseconds for the complete analysis, with a median value of only 860.5 ms. The difference between the median and the average is based on the fact that we have some outliers with a maximum processing time of 63,191 ms. The minimum processing time, which is often reached when analysing small websites without any JavaScript, is only 15 ms. The standard deviation is 4,291.8 ms. Thus we can conclude that usually the analysis of a website takes 0.1 to 6.4 seconds, when adding the standard deviation to the average processing time. The cumulative distribution function depicted in Figure 2 shows that 90% of all the websites could be analysed in less than 5.1 seconds, 97% still take less than 10 seconds and after 14.7 seconds we reach the 99% percentile. We consider this to be quite acceptable for a first unoptimized implementation.

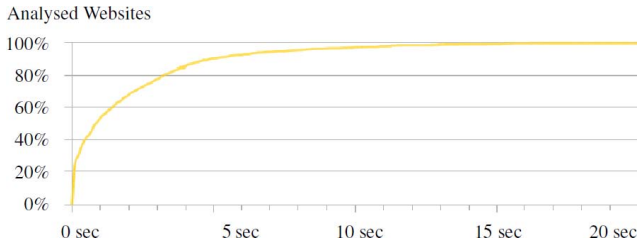

Figure 2: Cumulative distribution function of the processing time.

3.4 The Role of Obfuscation

We also examined the use of the `eval`, `write`, `writeln` and `unescape` when examining the mentioned alexa.com top websites, because these functions are often used by attackers to obfuscate their JavaScript code. It is a widespread belief that these functions, at least in combination with each other (for example `unescape` within an `eval` call), are only used on malicious websites and the occurrence of such combinations is a reliable indication for malicious intent. We wanted to examine this assumption and thus recorded the use of each of those functions. When regarding the results shown in Table 2, it becomes clear that the assumption does not hold in practice, since these functions are frequently used also on benign websites.

	# Calls	Avg. Calls/Site
Eval	4,387	4.39
Write/Writeln	38,432	38.43
Unescape	1,043	1.04
Unescape in Eval	2	0.00
Unescape in Write/Writeln	277	0.28

Table 2: Usage frequency of specific functions.

4. CONCLUSION

In this paper, we presented ADSandbox, a tool for analysing and detecting malicious websites. We employed a novel concept of a client-side JavaScript sandbox. Our solution takes advantage of the processing power available at the client and thus scales without problems. Furthermore, the system always provides an up-to-date analysis report to the end-user. The drawback for now is the high processing time on certain websites and the fact that our detection heuristics have to be refined and completed. The overhead could be mitigated by caching the analysis results and a central repository which each client could contact before performing an analysis on its own. And preliminary results show that we can already detect many malicious websites with a false-positive rate of zero. Furthermore our dynamic JavaScript analysis provides a flexible tool to understand what actions a script performs that might have been obfuscated and encoded several times. It is thus especially interesting for security researchers who often manually analyse malicious JavaScript.

When evaluating our system, we obtained some interesting results regarding the use of hidden IFrames and several JavaScript functions such as `eval`, `unescape` and `write`

that are counter-intuitive to common wisdom. What we have also observed is the presence of advanced obfuscation techniques that combine JavaScript with DOM objects, a technique that can not be detected by any automated static analysis (yet).

In the future, the framework should be easy to extend by new analyses which could, for example, aim on the detection of exploits against specific browser plug-ins. Further, we want to improve the performance of our tool through several optimisations. For example, we want to utilise the new version of Spidermonkey from Firefox 3, which is known to be much faster than the older version.

In addition, we plan to evaluate our tool further by analysing a larger number of websites and following links to do a deep scan on these sites. Another part of the evaluation is an analysis of methods for an attacker to detect our sandbox.

Acknowledgements

We would like to thank Martin Johns for valuable feedback on a previous version of this paper that substantially improved its presentation.

The authors gratefully acknowledge funding from *FIT-IT Trust in IT-Systems* (Austria) under the project TRUDIE (P820854), and from the *European Commission* under the projects WOMBAT and FORWARD.

5. REFERENCES

[1] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis. In *20th IFIP International Information Security Conference*, 2005.

[2] Benjamin Livshits, Weidong Cui. Spectator: Detection and Containment of JavaScriptWorms. In *USENIX Annual Technical Conference*, 2008.

[3] Christian Seifert, Ramon Steenson. Capture-Honeypot Client. `http://www.nz-honeynet.org/capture.html`, 2007.

[4] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A Safety-Oriented Platform for Web Applications. In *IEEE Symposium on Security and Privacy*, 2006.

[5] A. Dewald. Detection and prevention of malicious websites. Master's thesis, University of Mannheim, Department of Computer Science, 2009.

[6] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: mitigating heap-spraying code injection attacks. In *6th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2009.

[7] Google Inc. Safe Browsing for Firefox. `http://www.google.com/tools/firefox/safebrowsing`.

[8] James Newsome, Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[9] C. Krügel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[10] McAfee. SiteAdvisor. `http://www.siteadvisor.com`.

[11] Microsoft. IE8 Security Part III: SmartScreen Filter. `http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-iii-smartscreen-filter.aspx`, 2009.

[12] Mozilla Foundation. JSAPI Reference. `https://developer.mozilla.org/en/JSAPI_Reference`, 2009.

[13] Paruj Ratanaworabhan, Benjamin Livshits, Benjamin Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. Technical report, Microsoft Research Technical Report MSR-TR-2008-176, 2008.

[14] Samy. The Samy worm. `http://namb.la/popular`, 2005.

[15] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, and C. Yuan. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *USENIX LISA*, 2003.

[16] Yi-Min Wang and Doug Beck and Xuxian Jiang and Roussi Roussev. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2006.