A HYBRID SYSTEM FOR ANALYSIS AND DETECTION OF WEB-BASED CLIENT-SIDE MALICIOUS CODE

Vitor Monte Afonso
University of Campinas (UNICAMP)

André Ricardo Abed Grégio
Renato Archer IT Research Center (CTI)

Dario Simões Fernandes Filho
University of Campinas (UNICAMP)

Paulo Lício de Geus
University of Campinas (UNICAMP)

ABSTRACT

Malicious Web applications are a significant threat to computer security today. They are the main way through which attackers manage to install malware on end-user systems. In order to develop protection mechanisms for these threats, the attacks themselves must be carefully studied and understood. Several systems exist to analyze and detect malicious Web pages, but they leave much to be desired. In this article we propose a system that dynamically analyzes Web pages through a novel technique that combines machine-learning and signature-based identification to detect malicious Web behavior. Our tests show that the proposed system---Browser Attacks Detection (BroAD)---has a better detection rate when compared to existing state-of-the-art systems. The BroAD system also produces more information about the malicious code than those systems, therefore providing a better understanding of the Web malware.

KEYWORDS

WWW intrusion detection, JavaScript attacks, Information systems security, Dynamic analysis

1. INTRODUCTION

Internet users can currently access a broad range of services through their browsers, including financial operations, e-commerce and social networks. Those services dramatically change the way users interact with client-side applications, as they become more powerful. Complexity brings more vulnerabilities and attack points that make client-side applications very attractive to be exploited by attackers. Also, users tend to be much more negligent about security than server administrators, allowing attackers to reach a great number of victims through phishing, for example.

Usually, attackers need to execute some code in the victims' browser (client-side) to compromise their system. To accomplish that they host the deceptive code on site (server-side) and then lure the users to load malicious content by accessing the site (e.g., links in SPAM, infection of legitimate sites). Once the victim's browser loads the malicious content, the embedded code exploits some vulnerability present in the browser or in a browser component and installs a malware on the system.

The malicious code executed in the victim's browser is commonly developed using JavaScript and is used to steal personal information from the user, to take control of the browser or to exploit vulnerabilities and install malware on the system. These malware are used to send SPAM, to attack other systems over the network or to act as keyloggers. This kind of code is constantly evolving, forcing companies that develop defense solutions to study them so that a deeper understanding of the techniques employed and vulnerabilities exploited can be gathered.

There are some systems that provide automated analysis of Web malicious codes and that can indicate if they are suspicious or benign. However, those Web malware analysis systems fail at some point, either by being detected and evaded by the malware or by not being able to properly analyze the suspicious code.In this article we present a system that analyzes JavaScript attack code and monitors system calls at the O.S. (Operating System) level, so as to detect malicious sites while providing information about the code, as well as its interaction and changes on the exploited system, i.e. its behavior. The proposed system works as an extended high interaction honeyclient, since it does not use a browser emulator to load sites, instead capturing the exhibited behavior beyond JavaScript's realm. Thus, we developed a hybrid system in which a Windows library loaded in the Internet Explorer's memory performs the JavaScript monitoring and a Windows kernel driver collects O.S. system calls. Furthermore, to search for malicious behavior we apply four detection methods: JavaScript signatures, shellcode detection, system call signatures and anomaly detection of JavaScript code. The main contributions of our work are:

- We propose a novel architecture that unifies some of the existing detection approaches, extending the traditional analysis to the inspection of malicious Web codes followed by the monitoring of the changes that occurred in the target file system.
- We developed a four-way detection technique that allows our hybrid system to detect both drive-by download and information stealing attacks, filling a gap present in current state-of-the-art approaches;
- We deployed a prototype, tested actual Web malware and compared the results to current state-of-the-art publicly available systems, showing that our approach produces higher detection rates than the existing ones, while at the same time leveraging more information about the attack code behavior;
- We implemented an evasion code that demonstrates how easy it is to bypass other available systems.

The remaining of this article is divided as follows. Section 2 shows the types of attacks detected by the system, explains them and presents related work. Section 3 presents the components of the system, the tests performed are shown in Section 4 and Section 5 concludes the article.

2. BACKGROUND AND RELATED WORK

The existing approaches to analyze malicious sites or URLs have several limitations regarding the evaluation of the damage done ona compromised system. Either they are limited to JavaScript-based attacks, or they only perform the malicious code analysis if an exploit is successfully executed. In this section, we describe the most common client-side attacks suffered from a Web browser's perspective and discuss the advantages and disadvantages of the current Web malware analysis systems.

2.1 Common Client-Side Attacks

Although the most prevalent threat to browsers' security is drive-by exploit attacks, which are the main focus of the existing approaches to analyze and detect malicious code delivered through the Web, there is also the need to detect codes whose purpose is to steal personal information from the users, such as navigation history and cookies. In Section 3 we detail how our proposed system tackles both types of attacks, which are described below.

2.1.1 Drive-by Exploits

Drive-by exploit attacks are commonly used to install malware in the victim's systems. These attacks usually require three main steps to succeed, explained as follows. First of all, the attacker must write a malicious shellcode[1] in the user's browser memory region. To do it, the attacker has to deceive the user to visit a compromised Web page (e.g., by clicking on a link received on a SPAM or phishing e-mail[2]), which is specially crafted to force the browser to load the malicious code. Then, the browser is exploited through some vulnerability present in it or in one of its components, such as an ActiveX interpreter, a PDF reader

[1] Shellcode is a piece of code, usually written in machine code, which is executed after exploitation.

[2] Phishing e-mails are used to lure victims into providing some information, executing some software or accessing some link.

plugin, the Flash player etc. After the exploitation of a vulnerability occurs, the shellcode that was stored in the first step is executed, which can lead, for instance, to the download and execution of a malicious program.

2.1.2 Personal Data Extraction

The theft of personal data is a severe attack, as it disrupts the users' confidentiality and is one of the main motivations for online financial crimes. In this article, we focus on cookies and navigation history theft, but this type of attack could be launched against any personal information stored in the browser, such as information filled in Web forms. The theft of navigation history is a threat to users' privacy, whereas the cookies can be used to impersonate victims. Codes for both stealing navigation history and performing drive-by exploits may be injected from any Web page. Thus, when a user accesses this Web page, the malicious injected code sends his history information to an evil Web site configured to receive it. Conversely, the cookie stealing code needs to be executed in the context of the site whose cookies are required. Therefore, it usually involves a XSS (Cross-Site Scripting) attack.

2.2 Related Work

There are several research works being done regarding the detection and identification of malicious Web code. They differ mainly by the type of attacks they address, the kind of code analysis that is performed and whether they can be used to provide protection or are only informative. Informative systems such as (Cova, et al., 2010; Nazario, 2009; Seifert and Steenson, 2006) only analyze Web malicious code, i.e., they cannot be used to perform real time detection. In (Cova, et al., 2010), the authors present a system, JSand, which is used in a publicly available submission site (Wepawet, 2010). It analyzes JavaScript code, presents information about it and tells whether it is malicious, suspicious or benign. JSand uses machine learning techniques to detect malicious JavaScript behavior, similarly to our system proposed in this article. It uses a browser emulator to mimic the existence of any ActiveX object requested by the code. This feature is useful to extend the JSand system so as to allow for more samples to be analyzed. However, this opens up the possibility of JSand being easy to be detected and evaded. Also, JSand can only detect drive-by download attacks that make use of JavaScript.

PhoneyC (Nazario, 2009) also uses an emulator to process Web pages. It emulates certain vulnerabilities and is able to analyze malicious codes that use both VBScript and JavaScript languages, but it cannot analyze other types of attacks, such as the ones that use Java or Flash.

Differently from the aforementioned systems, Capture-HPC (Seifert and Steenson, 2006) is a high interaction honeyclient---a system prepared to visit and monitor potentially dangerous Web sites---that uses a full-featured Web browser, i.e., it does not emulate it. As our proposed system does, Capture-HPC uses a Windows Kernel Driver to capture the system calls performed by the browser, being ableto detect malicious code by analyzing them. However, it is only able to monitor attacks that were successful and if they used anomalous system calls.

Other systems are used to protect users while they are using the Web, instead of just providing information or identifying malicious codes. BLADE (Lu, et al., 2010) is a kernel extension used to block drive-by download attacks. It prevents binary files downloaded by the browser without user consent from being executed on the victim's system. However, this approach has the following shortcomings: it cannot detect attacks other than drive-by downloads, it cannot block executable script files and it cannot detect malicious codes that are executed directly from memory, without writing the binary to disc. In (Rieck, et al., 2010), the authors describe Cujo, a system that applies machine-learning techniques to detect and prevent attacks through the use of a proxy, while presenting a low overhead. However, it can only detect drive-by downloads that use the JavaScript language. Another system that uses a proxy for users' protection is the WebShield (Li, et al., 2011). It uses a browser emulator inside the proxy and an agent in the user's browser. The emulator executes the JavaScript code, checks for malicious behavior and blocks it if the code is detected as malicious. If no malicious behavior is detected, the content in the user's browser is updated. The limitation here is that WebShield cannot detect attacks that use other languages, such as Flash, Java and VBScript.

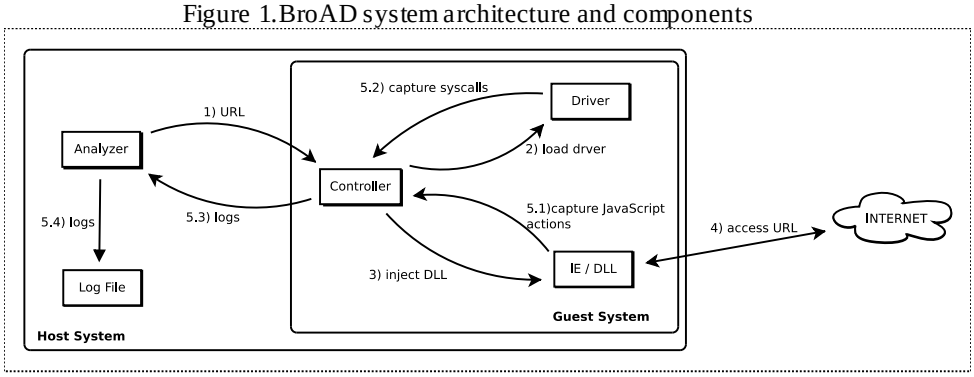
Apart from WebShield, all mentioned systems focus only on drive-by exploit attacks. There are other systems that focus on XSS-based attacks (e.g., attacks to steal information from the browser). For instance, in (Vogt, et al., 2007), the authors present a system that tracks the information flow inside the browser using dynamic data tainting and static analysis to stop information leak attacks.

3. SYSTEM DESCRIPTION

3.1 Architecture

In this section we present the components of our Web malware analysis system (BroAD – Browser Attacks Detection) in details, their interaction and what is their role in the system. BroAD dynamically analyzes malicious code samples in a virtual environment, so it is possible to restore the analysis system after it is compromised by a Web malware. There is a manager program (the analyzer) in the host system whose function is to control the entire analysis process and the programs that process the log files, and to identify the malicious behaviors. The DLL (Dynamic Link Library) that captures the JavaScript actions is located at the guest system, as well as a driver we developed to capture system calls and the controller program that starts the DLL and this driver. This architecture can be seen in the figure 1.

Figure 1.BroAD system architecture and components

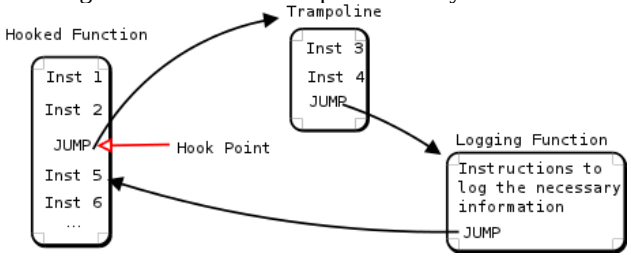


3.2 JavaScript Monitor

The DLL responsible for the monitoring of the JavaScript actions is loaded into the memory of the Internet Explorer by the controller and it modifies some parts of the Internet Explorer libraries. These modifications change the execution flow of some functions in a way that the logging functions are called before the original functions. This type of modification is known as hooking and the methodology used in this work is based on the one used by the Ultimate Deobfuscator(Chenette, 2008).

To determine the places where the hooks will be installed it is necessary to manually analyze the Internet Explorer libraries to find the correct location of the necessary information (memory addresses). Then, the DLL is generated with those found addresses, so it knows the places where the hooks have to be installed. These hooks are usually installed in the beginning of the functions and the hooking process works in the following way: first, the initial instructions after the hook point are copied to another memory area called trampoline. In the original function, these initial instructions are replaced by a jump instruction that redirects the execution flow to the trampoline area. In this area, another jump is placed right after the copied instructions. This time, the jump is destined to the function responsible for logging the required information, i.e., the parameters that were passed to the hooked function. A third jump is placed in the end of the logging function to redirect the execution flow back to the original function. This process can be seen in figure 2.

Figure 2. Modifications performed by the hooks



The information we capture from the monitored actions is related to string manipulation, dynamic code execution, DOM (Document Object Model) modification, ActiveX objects creation and invocation, array related functions, encoding and decoding functions, functions that manipulate HTTP requests and operations that manipulate system properties and user information. In malicious codes, the first three groups aforementioned are mostly used for obfuscation, i.e., to transform the code in such a way that makes the code vary hard to understand and manually analyze. The monitoring of ActiveX objects is an important step to detect their exploitation. Otherwise, arrays are usually used in head-spraying attacks where the shellcode is copied to the memory several times. Lastly, the HTTP requests and the manipulation of user information are important to detect the extraction of personal information.

A program that has access to the code in memory can easily detect the type of hooking that we use, but the JavaScript code cannot perform this detection, as it doesn't have direct access to the Internet Explorer memory. They could only be detected after the exploitation of some vulnerability in the client, but to accomplish this, the attacking code would already have executed the malicious actions, which our system captures and uses it to identify this code as malicious.

The hooking procedures that we deployed use hard-coded addresses of internal functions from some Internet Explorer (IE) libraries, which are the standard addresses used when these libraries are loaded. If there is a need to use our logging DLL together with other versions of these IE libraries, it is necessary to modify the hooked addresses and the way the information is captured (in cases of changes in parameters or internal structures).

3.3 Driver

The system call monitoring at the operating system's kernel level is accomplished with the use of a Windows Kernel Driver that modifies the SSDT (System Service Dispatch Table). The SSDT is a kernel structure where the memory addresses of the system native APIs are located. When the BroAD's controller program loads this driver, it changes specific SSDT addresses to assure that the driver's functions are called before certain native functions and that it logs the required information to allow the further detection. This driver is explained in more detail in (Grégio, et al., 2011).

3.4 System Configuration

The OS guest system is a Windows XP SP2 that uses the Internet Explorer 8 as the main browser and some additional installed applications that can be used by a Web malicious code, such as Adobe Reader, JRE and Flash player. Web malicious codes can exploit several different user applications. To be able to analyze all these different malicious codes, an analysis system needs to emulate those applications (in the case of low interaction honeyclients) or to have them installed (in the case of high interaction honeyclients). Some low interaction systems, such as PhoneyC, emulate certain vulnerable ActiveX objects whereas others emulate every ActiveX object requested by the code, such as the JSand system.

Figure 3. Code used to detect and bypass systems that emulate all ActiveX objects requested by the code

```
try{
      var obj = new ActiveXObject("FakeObject");
      /* inside analysis system, execute benign code */
} catch (e) {
      /* not in analysis system, execute malicious code */
}
```

The emulation of every ActiveX object helps in detecting Web codes that attack different applications, but this can turn the analysis system either detectable or evadable. For instance, the code presented in the figure 3 turns possible to detect and evade the analysis of a system that emulates all ActiveX objects. This was a decisive factor to us regarding the design of BroAD, so we chose not to do the things this way. While it is not possible to bypass our system by detecting emulated ActiveX objects, we have a flexibility downside: we have to constantly update the installed applications to be able to analyze the most recent malicious codes or the attacks that target applications that are not currently installed may not be detected. However, we opt to a higher cost of maintenance to guarantee high rates of detection and identification of malicious code, consequently increasing the security capability of BroAD related to future client-side protection features.

3.5 Information Provided

BroAD provides information about the JavaScript behavior, the system calls that were considered anomalous and the network traffic. The information related to the JavaScript execution includes the codes that were passed to the Internet Explorer's JavaScript parser, those that were used in functions that dynamically evaluate codes, the modifications to the DOM, the identified shellcodes and the ActiveX objects used. The network part includes the accessed IP addresses and the performed HTTP requests.

Among the existing systems we evaluate for this work, JSand/Wepawet is the one that provides more information about the analyzed Web malicious codes, but it doesn't show the changes to the DOM that are performed through modifications of the *innerHTML* property, which can also be used to evade the analysis. When Wepawet detects a malicious executable, it provides a link to the analysis report of this sample in the Anubis system (Anubis, 2010), a platform to analyze Windows executables. However, if the shellcode used during the attack performs more than just the download and execution of a piece of malware, Wepawet will not be able to report its performed actions. Otherwise, the other evaluated systems provide yet less information. CaptureHPC can only report the system calls that are considered anomalous whereas PhoneyC only provides information about the exploits and ActiveX objects used.

3.6 Detection

The detection of malicious behavior is performed by our proposed four-way technique – the signature verification of JavaScript actions, the shellcode detection, the signature verification of OS system calls and the anomaly-based detection, that will be detailed in the next sections.

3.6.1 JavaScript Signatures

To this work's end, we developed JavaScript signatures to identify malicious behavior, which are used to detect known attacks. These signatures are regular expressions whose purpose is to verify certain actions and their associated parameters. They are mainly used to detect information stealing attacks, as they must follow a determined sequence of actions. For example, one possible sequence of actions for an attack code to check if a user visited the Web site "www.test.com" is presented in the table 1. This table also presents the signature used to detect this attack. The parameter used with the *Send info* operation means that a HTTP request to a different domain is executed. Note that this is not the only way to perform this attack, mainly because there are several other possible ways to send the information to another Web page. This implies that it is necessary to create a signature to each different case, as in any signature-based system.

These signatures could also be used to identify which are the exploits used by the Web malicious code. This feature is not implemented yet in our current system and we let it as a future work. However, if our classifier identifies a Web page as malicious, the JavaScript signatures could be used to identify which vulnerability is exploited based on the interaction with ActiveX objects.

Table 1.Regular expressions for detection of navigation history stealing

Operation	Regex for Operation	Regex for Parameter
Create a link (tag *a*) pointing to "www.test.com"	CALL DOM.CREATE_ELEMENT	^[aA]$
Insert the link in the page	CALL DOM.INSERTELEMENTET	-
Get the color of the link	GET COLOR	-
Based on the color it is possible to know if the site is visited. If so, send the information to "www.evil.com"	CALL XMLHTTP.OPEN	^(?!${SOURCE}).*

3.6.2 Shellcode Detection

There is a step to verify for the presence of shellcodes. The strings that are used as a parameter or as a result of some selected function calls are examined and then they are verified by their mime-type. If the string is considered suspicious, i.e., if the mime-type does not contain the string *text*, it is verified using the Libemu (Baecher and Koetter, 2008) tool. If Libemu detects a shellcode, the Web page being analyzed is considered malicious.

3.6.3 System Call Signatures

The OS system call signatures are used to detect actions seen in the system but that were not supposed to be performed without the user consent. As the BroAD analysis system runs without any user interaction (fully automated), all of the monitored system calls performed are suspicious as, they are a result of the browser accessing a Web page. Hence, if a process is started from the temporary page of the Internet Explorer, it is considered malicious, for instance.

The system call signatures are created as regular expressions whose purpose is to define which set of system calls are accepted as inoffensive and which of them configure a malicious behavior. Thus, those signatures can detect any successful attack that results in the installation of a malware sample or that results in changes on the file system that should not be performed automatically by the browser. This is useful mainly in cases in which JavaScript is not used in the exploitation phase, such as attacks by Java, VBScript, Flash and other language codes.

3.6.4 Machine Learning

To perform the anomaly-based detection we use eight features that were extracted from the JavaScript logged actions. Most of the features used where based in the features three to nine presented in (Cova, et al., 2010). Moreover, we added another feature to represent the sum of the size of the strings that were identified as possible shellcodes. The features include the number and size of string definitions and strings allocated in arrays, the number of dynamically evaluated code and DOM modifications, the size of dynamically evaluated code, the number and size of possible shellcode strings, the number of ActiveX objects created and the size of the parameters passed to ActiveX objects' methods.

Drive-by exploit attacks usually make use of the JavaScript language to load a shellcode into the memory and to trigger a vulnerability that is present in the browser or its applications. Even when this kind of attack is not entirely completed, it is still possible to detect it due to our attribute extraction if at least the shellcode is loaded successfully. We use the Weka framework (Hall, 2009) to build the classifier that is responsible to perform the detection of Web malicious code.To this end, we chose the metaclassifier ThresholdSelection and the RandomForest (Breiman, 2001) algorithm, as it produced the best results in the tests (Section 4).

4. TESTS AND RESULTS

To show the effectiveness of the BroAD system, we compared it to thethree more popular Web malware analysis systems – JSand, PhoneyC and CaptureHPC. We used two datasets of HTML files and URLs; one of themconsists of malicious HTML files and the other one consists of benignURLs. The malicious dataset was obtained from the VxHeaven database (VxHeavens, 2010) and from URLs listed as malicious by the Malware Domain Listsite (Malware Domain List, 2011). This dataset has 2389 samples. The benign dataset wasobtained from the 12 thousand first entries of the Alexa (2011) TOP sites. These URLs were verified on the Google safe browsingservice and the URLs that were marked as containing malicious contentwere removed from the dataset, resulting in 10096 benign samples.

The datasets were separated in two subsets: training and test. The training and the test malicious datasets contained 717 and 1672 samples, respectively. The training and the test benign data sets contained 3077 and 7019 samples, respectively. The 10-foldcross validation of the training data resulted in 0,91% of false positives (benign classified as malware) and18,41% of false negatives (malware not detected).

The detection results of our tests using the aforementioned data sets and the four evaluated systems (BroAD and the other three) are presented in the Table 2. This table also presents the number of false-positives, false-negatives, true-positives and true-negatives produced by each system. To this end, the JSand samples considered as suspicious were grouped together with the malicious ones.

From the true-positives rate, the anomaly detection step was responsible for 76,50%, the JavaScript signatures detection were responsible for 1,73% and the OS system call for 16,51%. It is worth to note that the percentages do not sum 81,10% (the true-positive rate of BroAD), as more than one of our methods could have detected the Web code as malicious. Moreover, the accuracy of a test can be measured by the harmonic mean, or F-measure, which considers the test's precision and recall. If we take a closer look on the Table 2, we can observe that BroAD's F-measure is 89.4%, which is more than twice of the score calculated for the second best system (JSand) and consequently demonstrates the quality of our approach.

Table 2. Detection results using the benign and malicious datasets to evaluate our system and compare it to JSand, PhoneyC and Capture-HPC, where M represents the amount of samples that were considered malicious, S the suspicious, B the benign and E the error. The false-positives (FP), false-negatives (FN), true-positives (TP), true-negatives (TN) and the harmonic mean (F-measure) are shown in the " General" column.

Dataset	BENIGN				MALICIOUS				GENERAL (%)					
Detected as	**M**	**S**	**B**	**E**	**M**	**S**	**B**	**E**	**FP**	**FN**	**TP**	**TN**	**E**	**F-measure**
BroAD	22	0	6997	0	1356	0	316	0	0,31	18,90	81,10	99,69	0	***89.40***
JSand	1	126	5881	1011	335	116	1211	10	1,81	72,43	26,97	83,79	11,95	***42.08***
PhoneyC	0	0	5943	1076	180	0	1368	124	0	81,82	10,77	84,67	13,81	***20.83***
CaptureHPC	10	0	6117	892	96	0	1522	54	0,14	91,03	5,74	87,15	10,89	***11.18***

5. CONCLUSION

Analyzing malicious Web pages is an important task to the development of counter-measures and protection systems to modern browsers, which are valuable targets to attackers that intend to install malware or steal information from the users. However, the existing systems that perform such kind of analysis are limited when regarding the detection and analysis of malicious Web code. To address those limitations, we presented a novel approach to detect malicious Web pages. Our proposed system, namely BroAD, combines the monitoring of O.S. system calls and the analysis of JavaScript behavior on a four-way detection scheme that applies machine-learning and signature-based techniques. We performed tests (training and classification) with more than 12,000 URLs (benign and malicious) obtained 'in the wild' and BroAD's results presented better overall detection and accuracy rates when compared to state-of-the-art Web malware analysis systems. Future works include the analysis of other kind of languages (VBScript and Java), as well as data generation to input into protection mechanisms, aiming real time detection of malicious Web codes.

REFERENCES

Cova, M. et al., 2010. Detection and analysis of drive-by-download attacks and malicious javascript code. *Proceedings of the 19th International Conference on World wide web.* New York, USA, pp. 281-290.

Li, Z. et al., 2011. WebShield: Enabling Various Web Defense Techniques without Client Side Modifications. *Annual Network and Distributed Systems Security Symposium (NDSS).*San Diego, USA.

Chenette, S., 2008. The ultimate deobfuscator.*Proceedings of the ToorConX Conference.* Seattle, USA.

Lu, L. et al., 2010, Blade: An attack-agnostic approach for preventing drive-by malware infections. *Proceedings of the 17th ACM Conference on Computer and Communications Security*. Chicago, USA, pp. 440-450.

Vogt, P. et al., 2007, Cross-site scripting prevention with dynamic data tainting and static analysis.*Proceedings of the Network and Distributed System Security Symposium*.San Diego, USA.

Nazario, J., 2009, Phoneyc: A virtual client honeypot. *Proceedings of the 2nd USENIX Conference on Large-Scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*. Boston, USA, pp. 6-6.

Rieck, K. et al., 2010, Cujo: efficient detection and prevention of drive-by-download attacks. *Proceedings of the 26th Annual Computer Security Applications Conference*.Orlando, USA, pp 31-39.

Seifert, C. and Steenson, R, 2006, Capture – honeypot client (capture-hpc).

Hall, M. et al., 2009. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1), pp 10-18.

Breiman, L., 2001. Random Forests. Machine Learning, 45 (1), pp 5-32.

Alexa, 2011.Alexa- the web information company. [online] Available at: <http://www.alexa.com/> [Accessed Apr/2011]

Malware Domain List,2011. [online] Available at: <http://www.malwaredomainlist.com/> [Accessed Sep/2010]

VxHeavens, 2010.[online] Available at: <http://vx.netlux.org> [Accessed Sep/2010]

Wepawet, 2010.[online] Available at: <http://wepawet.cs.ucsb.edu/> [Accessed May/2011]

Baecher, P. and Koetter, M., 2008. Libemu-x86 shellcode detection and emulation. [online] Available at: <http://libemu.carnivore.it> [Accessed May/2011]

Anubis, 2010. Analyzing Unknown Binaries. [online] Available at: <http://anubis.iseclab.org> [Accessed Jun/2011]

Grégio, A. et al., 2011. Behavioral analysis of malicious code through network traffic and system call monitoring. *Proceedings of SPIE Volume: 8059*. Orlando, USA