SafeBrowsingCloud: Detecting Drive-by-Downloads Attack Using Cloud Computing Environment

Haibo Zhang, Chaoshun Zuo, Shanqing Guo, Lizhen Cui, and Jun Chen

Shandong University,
Shunhua Road, Jinan, P.R. China
{guoshanqing,clz,jchen}@sdu.edu.cn,
gsq_cy@163.com, guosq2002@hotmail.com

Abstract. Drive-by downloads attack has become the primary attack vehicle for malware distribution in recent years. One existing method of detecting drive-by download attacks is using static analysis technique. However, static detection methods are vulnerable to sophisticated obfuscation and cloaking. Dynamic detection methods are proposed to overcome the shortcomings of static analysis techniques and can get a higher detection rate. But dynamic anomaly detection methods are typically resource intensive and introduce high time overhead. To improve performance of dynamic detection techniques, we designed SafeBrowingCloud, a system based on apache S4, a distributed computing platform. And the system is deployed at edge router. SafeBrowingCloud analyzes network traffic, executes webpages in firefox with modified javascript engine, abstracts javascript strings and detects shellcode with three shellcode detection methods to find malicious web pages. Experimental results show efficiency of the proposed system with the high-speed network traffic.

Keywords: Drive-by download, Apache S4, Shellcode, Cloud.

1 Introduction

Drive-by download attack is one of the main vectors used to spread malware. In a drive-by download attack, an attacker first presents a web page including malicious code which tries to exploit a vulnerability in the victim's browser or in a browser plugin. When a web visitor browses the malicious web page, the injected code instructs the victim's computer to download and install malicious software. The installed malware often enables an attacker to control a user's computer and steal sensitive information.

To protect users from drive-by download attack while browsing the Internet, several methods for detecting malicious web pages have been proposed. One existing malicious web page detection technique involves the use of what are known as a static detection technique, which uses data mining and machine learning technique[1,5,7]. The detection time of static detection methods is faster than that of dynamic detection methods but the detection rate is lower due

R.C.-H. Hsu and W. Shangguang (Eds.): IOV 2014, LNCS 8662, pp. 292–303, 2014.
© Springer International Publishing Switzerland 2014

to sophisticated obfuscation and cloaking[3]. To overcome the shortcomings of static detection method, dynamic detection technique is proposed[13], which run the scripts associated with web pages on a virtual machine to detect if the page is malicious. The main idea of these systems is to monitor a computer system for anomalous changes during the rendering of a web page such as changes to the file system, registry information or creation of processes. The dynamic detection method can get a higher detection rate than the static method. [2,9]. Dynamic detection technique can conquer some drawbacks of the static detection and the detection rate is higher than static method. However, most dynamic techniques are typically resource intensive and introduce high time overhead, making these approaches difficult to deploy as online detectors [3].Therefore, dynamic detection is not very applicable to large scale, real time classification[2], especially to complete the analysis of the flow of hundreds of megabytes by intrusion detection equipment[12], not to mention on Gigabit network traffic.

Traditional methods either distribute a security software for users to install or analyze webpages offline. Not every user would like to install a third party software. And offline analysis can't achieve realtime protection for drive-by download attack. To protect hosts from drive-by download attack, we would implement protection at edge router.

To protect from Drive-by downloads attack at edge router, the system should be able to analyze high speed network traffic, get http requests, load the webpages and detect shellcode when javascript executing. The scale of a network can change from several to thousands.One scalable system is necessary to accomplish above jobs, which can adjust its process capacity according to the network's scalability. To detect malious webpage as soon as possible,the system should have good real-time processing capacity. To this end, we implemented our method on Apache S4, a distributed stream processing platform.

Paper Organization. The rest of this paper is organized as follows. Section 2 presents our system design and implementation. Section 3 evaluates SafeBrowsingCloud's performance. Section 4 reviews related works. Section 5 concludes.

2 System

In this section, we proposed SafeBrowsingCloud, a cloud platform used for filtering web-based attack as users visit a web page. We intend the system to act as a first layer of defence against web-based attack.

We show the overall architecture of SafeBrowsingCloud in Figure 1. There are five kinds of nodes in SafeBrowsingCloud:the adapter node, the processing node, the zookeeper node, storage node and idle node. The adapter nodes are responsible for receiving router network traffic and dispatching http packets to processing nodes. The processing nodes hold the PEs to process data. The zookeeper node monitors the status of the system, reports node failures, and records system running statistics. The idle nodes are running as stand-by nodes. Once a node happens to crash, an idle node would take the place of it and take over its function. The storage node stores blacklist, whitelist and third

party engine database. Here the third party engine database stores malicious and legitimate URLs provided by third party engine,like google safe browsing service and so on.This database would update periodically.

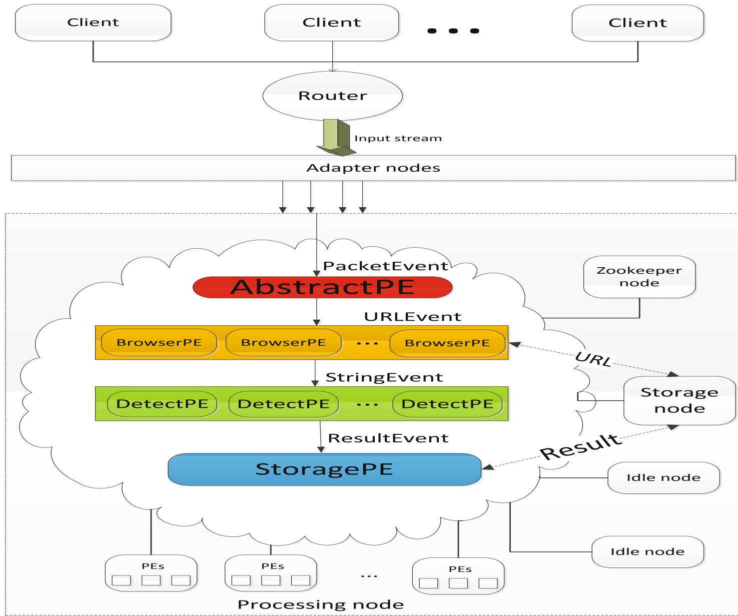

Fig. 1. System Architecture

The processing node processes data by internal PEs. We design four kinds of PE, including AbstractPE, BrowserPE, DetectPE and StoragePE.Every kind of PE is responsible for processing one kind of Event. And the Events are designed as Table 1. The AbstractPE analyzes network data packets, abstracts URL from http request and emits URLEvent which includes the URL. BrowserPE receives a URLEvent, visits each URL by firefox with modified spidermonkey, to collect the strings produced by running the javascript embedded in the web pages, and emit StringEvents. The DetectPE receives a StringEvent, detects if the string is a shellcode and emits ResultEvent. The StoragePE receives a ResultEvent and updates the database in the StoragePE node. SafeBrowsingCloud's final decision is returned to the party that submitted the URL; they can then take appropriate action based on their application, such as displaying a warning that users can click through, or deleting the content that contained the URL entirely. We now give an overview of each PE in this workflow.

2.1 AbstractPE

The AbstractPE is mainly responsible for analyzing network traffic to abstract URLs. We abstract the TCP socket (source IP:source port,destination IP:

Table 1. Events in SafeBrowsingCloud

Event type	Key-Value pair	Event entity
PacketEvent	null	byte array including network data packet
URLEvent	"url",http request url	URL,socket,timestamp
StringEvent	"url",url that produced the string	URL,javascript string
ResultEvent	null	URL,malicious or legitimate,timestamp

destination port), request time and url to record users' http requests history. The url would be submitted to BrowserPE for analyzing the webpage if malicious.

2.2 BrowserPE

For a drive-by attack to succeed, it is important that the shellcode is loaded into memory and interpreted as valid x86 instructions. In javascript,the only way to storing shellcode is by using a string variable.

To detect the shellcode that a malicious script might construct on the heap, we have to keep track of all string variables that the program allocates[2]. To this end, we modified spidermonkey, the JavaScript engine used by Firefox. In javascript, there are mainly three kinds of string operation, including string initialization, string concatenating and string spliting. As strings in JavaScript are immutable, all three kinds of manipulation would lead to string being created in a new memory area. We just need to add code to all points where a new string is created in spidermonkey. To implement string operations, spidermonkey organizes some string classes into a single inheritance hierarchy. For each string in JavaScript, there is a corresponding instance of JSString, the base string class in spidermonkey. Each concrete class corresponds to a particular implementation of JSString. For instance, JSRope is optimized to represent concatenated strings, and JSDependentString to represent substrings. The SpiderMonkey API provides corresponding functions that can be used to create instance of concrete class of JSString. For example, JSNewStringCopyN and JSNewStringCopyZ are used to create instances of JSFixedString. We would track these functions and abstract created strings. Besides, we need to tell which web page produces the string. In spidermonkey, the cx variable of JSContext type contains the html location information.Once a string is created, the string along with the web page url would be both put in a StringEvent.

The BrowserPE first query the database of the storage node if the url has been detected before. If the url has existed in the database, then the BrowserPE does nothing. Otherwise, it would execute the web page with received url, trace the javascript string variables and put them in StringEvent with the url.

2.3 DetectPE

In computer security, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. To evade exploit detection, poly-

morphic or metamorphic technique is applied to shellcode, which creates a polymorphic shellcode.

Shellcode detection methods can be classified by the shellcode part they detect[4]. There are NOP-sled detection, decryption routine detection, shellcode payload detection and return address zone detection methods.Every detection method has its advantage and disadvantage. NOP-sled detection method has a higher detection speed than other methods. However, NOP-sled may be missing in advanced exploit code and NOP-sled detection method would become invalid. Decryption routine detection method can detect polymorphic shellcode with higher detection rate than other methods. In consideration of decryption routine detection method focusing on decryption routine detection, it would perform badly on typical common shellcode which doesn't have decryption routine.Shellcode payload detection method would have a higher detection rate on detecting typical common shellcode while it's not able to detect polymorphic shellcode well.Given the three methods' feature, we adopt all of them to detect shellcode.

The DetectPE is just a general term of PE that detect shellcode in SafeBrowsingCloud. We designed three kinds of DetectPE, including NOPSledPE, DecryptionPE and PayloadPE to implement above three shellcode detection methods respectively.

The NOPSledPE implements the Racewalk[4] algorithm proposed by Dennis Gamayunov et al. to detect shellcode by NOP-sled. Racewalk proposed a novel approach for NOP-sled detection using IA-32 instruction frequency analysis and SVM-based classification. The method is based on the fact that Intel instructions frequency characteristics for NOP-zones is different from normal data. It first analyzes sequence using cache, pruning techniques and Disassembly prefix tree to reduce computational complexity and then implements SVM classification to reduce the false positives rate.Analysis rate over 600 Mbit/sec using single CPU core allows to use Racewalk algorithm for gigabit network.

The DecryptionPE implements the method proposed by Qinghua Zhang et al. to detect self-decrypting exploit code[14]. The method detects the presence of a decryption routine,which is a characteristic of polymorphic shellcode. It uses static analysis and emulated instruction execution techniques to find the starting location and identify the instructions of the polymorphic exploit code. In addition, it can detect polymorphic exploit code that is self-modifying and that do not have a NOP-sled, which static analysis has previously been unable to detect. Its detection speed is roughly linear to stings' length and amount.The current implementation achieves a speed more than 10M/s.

The PayloadPE implements the SigFree[12], a signature-free buffer overflow attack blocker proposed by Xinran Wang et al.. SigFree first blindly dissembles and extracts instruction sequences from a request. It then applies a novel technique called code abstraction, which uses data flow anomaly to prune useless instructions in an instruction sequence. Finally it compares the number of useful instructions to a threshold to determine if this instruction sequence contains

code.SigFree is signature free, thus it can block new and unknown buffer overflow attacks.

In consideration of three DetectPEs' detection speed and shellcode structure, one coming string would be first detected by NOPSledPE. If NOPSledPE determines the string malicious, result would be sent to StoragePE. Otherwise, the string is submitted both to DecryptionPE and PayloadPE to be detected.

There are a large number of DetectPE instances in one processing node to detect shellcode.The DetectPE instances will be automatically removed from the cache once a fixed duration has elapsed after the PEs creation, or last access.

2.4 StoragePE

StoragePE processes the ResultEvent ,which contains the url, malicious or legitimate and the timestamp when the string is detected.The url is default stored in whitelist. Once a malicious ResultEvent is received, the url would be put in blacklist. The StoragePE would detect the blacklist and whitelist periodically. If one record's timestamp exceeds one threshold, the StoragePE would emit an URLEvent with the url to BrowserPE to update the record.

3 Evaluation

This section discusses how we evaluated our prototype as well as the experimental results. The evaluation was carried out in three parts. First we described our experimental setting. Second, we evaluated our system performance, including processing speed and system overhead. Third, we evaluated our system for false positives by accessing a large number of popular benign web pages and we used our system on pages that launch drive-by downloads and evaluated the detection effectiveness.

3.1 Experimental Setting

As shown in figure 1, our datasets come from personal computers' browsing records distributed in our campus. We captured the network packets and then abstracted urls. We make use of firefox to execute one url and modified firefox's javascript engine spidermonkey. Once a string object is produced in spidermoneky, it will be sent to SafeBrowsingCloud for being detected. In our experiment, there is an important parameter, url count that are executed at one time in firefox. We set a threshold of tab count in consideration of local servers' network environment and firefox's procesing capacity. In firefox, we write one extension to monitoring tabs behavior. The extension would notify SafeBrowsingCloud the tab count while tabs count changes and close the tab which has loaded over the url. We implemented the experiment of SafeBrowsingCloud for a small deployment consisting of 5 instances on the infrastructure in our network center. The SafeBrowsingCloud runs on the same infrastructure with four Intel Xeon 3.06 GHz CPU and 2 GB RAM. The computer is connected to a university campus network through 100 Mbps Ethernet;it runs Centos 6.4 Linux with kernel version 2.6.32.

3.2 Processing Performance

In this part, we evaluated the processing capacity of SafeBrowsingCloud. The evaluation was carried out in two parts. First, we tested processing speed. Second, we tested the whole system's overhead.

Processing Speed. To evaluate processing speed of the SafeBrowsingCloud system, we deployed the processing cluster with one to five processing nodes and recorded the maximum number of processed urls per hour respectively. And the result is shown in figure 2. It can be observed that the processing speed of SafeBrowsingCloud is approximately linear to the number of processing nodes, which demonstrated the scalability of SafeBrowsingCloud. To evaluate how many processing nodes are needed at least to process the campus network traffic, we analyzed the http request during one week. After one week analysis, we found at peak time the whole campus produced 11,000 distinct url per hour while it produced less than one thousand per hour when clients are not active. From figure 2, we can see with five processing nodes,the system is able to process the url in time at peak time. If http request speed becomes higher, we just need to add processing nodes. We only detect once for duplicated request. The system would detect one url again once duration between now and last processed time exceeds one threshold at idle time.

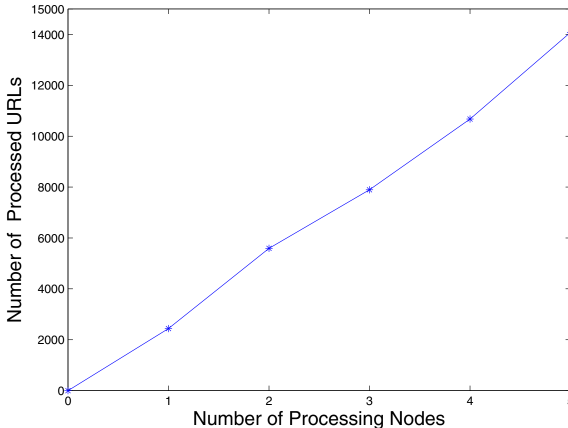

Fig. 2. Processing speed

Experiment Overhead. To evaluate the performance of SafeBrowsingCloud, we also tested its overhead, including cpu, memory and throughout of every processing node. The results are shown in figure 3, figure 4 and figure 5 respectively. It can be observed that the overhead changes of five processing nodes is similar, which demonstrated the SafeBrowsingCloud is decentralized and every processing node is equal. In addition, we periodically dump system running statistics

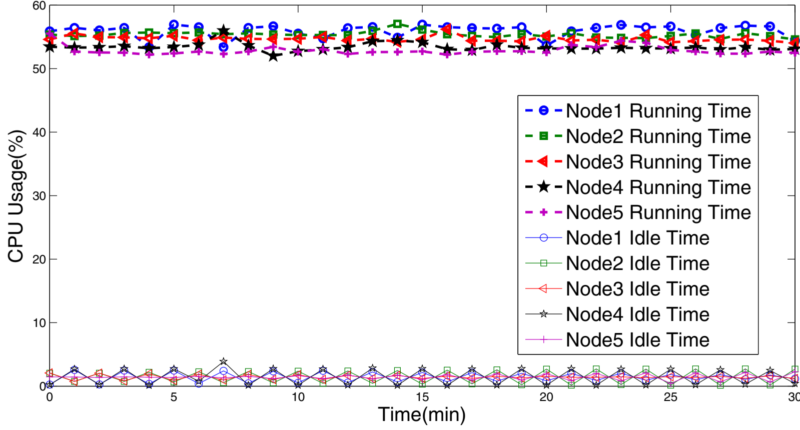

Fig. 3. CPU overhead

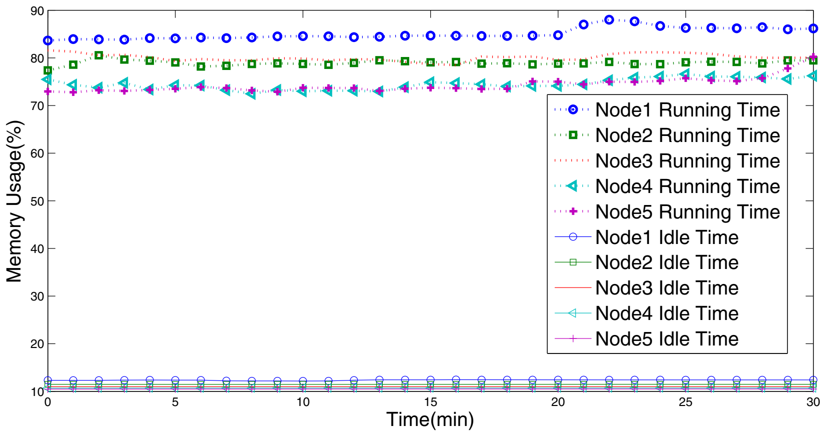

Fig. 4. Memory overhead

to logs using the metrics library, which offers an efficient way to gather such information and relies on statistical techniques to minimize memory consumption. During the running time, we found not events loss, which means all packets get processed and every javascript string is detected in time by our detecting classifiers.Observed from figure 3 and figure 4, memory and CPU is not made full use of. It is the firefoxPE that leads to it. As described before, we set a threshold in firefox to decide whether to request a url. The threshold is determined by network environment and firefox's processing capacity. With better network,the processing node would be able to process more urls per hour and the whole SafeBrowsingCloud would get a higher processing speed.

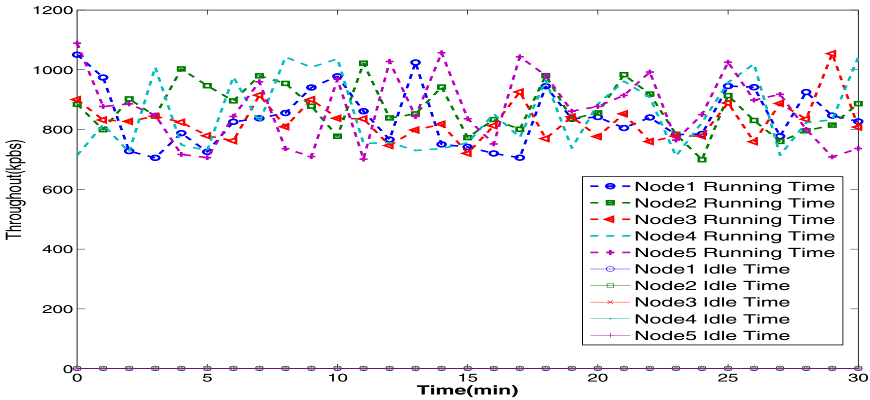

Fig. 5. Network throughout

3.3 Detecting Accuracy

False Positive Evaluation. In the context of our system, a false positive is a page that is detected as malicious without actually loading shellcode to memory. To evaluate the likelihood of false positives, we extended our prototype system to visit a list of k = 1000 known, benign pages. These pages were taken from the Alexa ranking of global top-sites, and simply consisted of the top k pages. We consider this to be a realistic test set that reflects a wide range of web applications and categories of content. For the batch evaluation of URLs, we implemented a Firefox extension that visits all URLs provided in a file. After last URL loaded over, the extension automatically visits the next URL in the list.Our prototype did not produce any false positives for this dataset. This might look suspicious at a first glance: The x86 instruction set is known to be densely packed, thus, almost any sequence of bytes makes up valid instructions. However, one has to consider the fact that JavaScript uses 16-bit Unicode characters to store text. That is, even if a given sequence of ASCII characters results in a valid x86 instruction most of the time, the JavaScript representation of the same characters most likely does not, since every other byte would contain the value 0x0. Of course, an attacker can encode the shellcode appropriately. However, benign pages typically do not contain strings that map to valid instruction sequences.

Detection Effectiveness. In a next step, we evaluated the capabilities of our technique to identify drive-by attacks that rely on shellcode to perform their malicious actions. To this end, we first collected 50 plain malicious shellcodes from the Internet and the Metasploit Framework, a powerful open source framework for the construction and execution of exploits. Based on the plain shellcodes, we generated 1000 polymorphic shellcodes by using Metaspoit Framwork and two off-the-shelf polymorphic engines:ADMmutate and Clet. We then create

1000 javascript codes that generate these malicious shellcodes at runtime. With above malicious javascript codes we created 1000 malicious webpages. We put these malicious webpages on our internal Web server. We used the aforementioned Firefox extension visiting all the URLs provided in a filed. And the our system correctly labeled them as malicious urls.

4 Related Work

Since drive-by download attack came out, several methods have been proposed to mitigate the threat. Efficiency and effectiveness are two main target in detection of drive-by download attack. Given different efficiency, these methods can be used in offline solutions or realtime solutions.

4.1 Off-line Solutions

Webpage content analysis is one technique used in off-line solutions to detect malicious webpages. Seifert et al.[10] proposed high interaction client honeypots to monitor the entire operating system, including a web browser, using a virtual machine. When a malicious page is loaded in browser, unexpected system state changes would happen.

To improve efficiency of off-line solutions, Seifert et al.[11] present an algorithmic approach by batch processing multiple web pages simultaneously. When multiple pages is found in a batch, it would use static analysis to find benign web pages and eliminate them first, thus reducing the number of pages to be detected.

Off-line solutions tend to be resource intensive and introduce long latency. Thus, off-line solutions are not suitable for realtime detection of drive-by download attacks.

4.2 Realtime Solutions

Now realtime solutions are mainly embedded in the web browser.

Lu et al.[8] proposed a BLADE system, that creates a non-executable sandbox to prevent any binary file to execute without explicit user intervention.

Jayasinghe et al.[6] proposed a novel approach to detect drive-by downloads in web browser environment using low resource dynamic analysis. By dynamically monitoring the bytecode stream generated by a web browser during rendering, the approach is able to detect drive-by download attack at runtime.

Ratanaworabhan et al.[9] proposed NOZZLE that detect shellcode in the memory heap using NOP sled detection to detect drive-by download attacks.

Above realtime detection techniques can improve efficiency of detection of drive-by download attacks. Realtime solutions are mainly embedded in the web browser to get realtime information like javascript objects, files, bytecode stream and so on. They make use of the realtime information to detect malicious page. Our solution is similar to NOZZLE, which detects drive-by download attacks by

detecting shellcode. Given that not all users would like to accept a third party browser which would introduce latency inevitably, we implemented our solutions at edge router. To process high speed network traffic, we designed our solutions on apache S4, a distributed computing platform.It avoids duplicate detection and detects malicious pages quickly.A page detection database is built that can be used by others like a http proxy to intercept drive-by download attacks.

5 Conclusions

In this paper, we designed SafeBrowsingCloud, a system used to detect drive-by download attack. In doing so, we analyzed network traffic, modified spidermonkey to abstract javascript strings and designed a shellcode detector. SafeBrowsingCloud is implemented based on apache S4, a distributed computing platform. The experiment demonstrates that our system can detect drive-by download attack with high efficiency and provide a safe browsing service at edge router.

References

1. Bannur, S.N., Saul, L.K., Savage, S.: Judging a site by its content: learning the textual, structural, and visual features of malicious web pages. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, pp. 1–10. ACM (2011)
2. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 88–106. Springer, Heidelberg (2009)
3. Eshete, B., Villafiorita, A., Weldemariam, K.: Malicious website detection: Effectiveness and efficiency issues. In: 2011 First SysSec Workshop (SysSec), pp. 123–126. IEEE (2011)
4. Gamayunov, D., Quan, N., Sakharov, F., Toroshchin, E.: Racewalk: fast instruction frequency analysis and classification for shellcode detection in network flow. In: 2009 European Conference on Computer Network Defense (EC2ND), pp. 4–12. IEEE (2009)
5. Hou, Y.-T., Chang, Y., Chen, T., Laih, C.-S., Chen, C.-M.: Malicious web content detection by machine learning. Expert Systems with Applications 37(1), 55–60 (2010)
6. Jayasinghe, G.K., Shane Culpepper, J., Bertok, P.: Efficient and effective realtime prediction of drive-by download attacks. Journal of Network and Computer Applications 38, 135–149 (2014)
7. Likarish, P., Jung, E., Jo, I.: Obfuscated malicious javascript detection using classification techniques. In: 2009 4th International Conference on Malicious and Unwanted Software (MALWARE), pp. 47–54. IEEE (2009)
8. Lu, L., Yegneswaran, V., Porras, P., Lee, W.: Blade: an attack-agnostic approach for preventing drive-by malware infections. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 440–450. ACM (2010)
9. Ratanaworabhan, P., Livshits, V.B., Zorn, B.G.: Nozzle: A defense against heap-spraying code injection attacks. In: USENIX Security Symposium, pp. 169–186 (2009)

10. Seifert, C., Komisarczuk, P., Welch, I.: True positive cost curve: A cost-based evaluation method for high-interaction client honeypots. In: Third International Conference on Emerging Security Information, Systems and Technologies, SECURWARE 2009, pp. 63–69. IEEE (2009)
11. Seifert, C., Welch, I., Komisarczuk, P.: Application of divide-and-conquer algorithm paradigm to improve the detection speed of high interaction client honeypots. In: Proceedings of the 2008 ACM Symposium on Applied Computing, pp. 1426–1432. ACM (2008)
12. Wang, X., Pan, C.-C., Liu, P., Zhu, S.: Sigfree: A signature-free buffer overflow attack blocker. IEEE Transactions on Dependable and Secure Computing 7(1), 65–79 (2010)
13. Wang, Y.-M., Niu, Y., Chen, H., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Strider honeymonkeys: Active, client-side honeypots for finding malicious websites (2007), http://research.microsoft.com/users/shuochen/HM.PDF
14. Zhang, Q., Reeves, D.S., Ning, P., Iyer, S.P.: Analyzing network traffic to detect self-decrypting exploit code. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, pp. 4–12. ACM (2007)