journal homepage: www.sabauni.net/ojs

Saba Journal of Information Technology and Networking (SJITN)

Article

Detecting Polymorphic No-Operations in Shellcode Based on Mining Techniques

Tawfiq S. Barhoom * , Fady R. Alkhateeb

Information Technology, Islamic University, Gaza, Palestine

Article info

Article history:

Accepted: April,2017

Keywords:

Mining,
Shellcode,
Buffer Overflow,
No-Operations,
Polymorphic.

Abstract

Shellcode acts as a weapon to perform Buffer Overflow (BOF), which is ranked as the most dangerous vulnerability. It consists of three sections that always transform their parts to be a Polymorphic Shellcode. Solutions available from Intrusion Detection Systems (IDS) still depend on the signature. Also, solutions that use data mining depend on Shellcodes with the factor of including payloads and not getting the high results, so polymorphic and unknown Shellcodes could not be detected. We proposed a new solution using a data mining classification technique on special features extracted which depends on the operation code of no operation instructions; which can classify the packets on the transport layer of the network as clean or buffer overflow Shellcode attack. This solution can detect unseen Shellcodes.

A dataset generated for malicious packets consists of 500,000 files from Metasploit No-Operation engines and 72,000 files of a clean dataset from various types of data. By applying different classification methods on the dataset which include selected features we specified and evaluating them by evaluation metrics; it showed that the solution has achieved high accuracy results with a 94% rate. In contrast, signature based on SNORT IDS detects only 50.02% of polymorphic Shellcodes in the experiment that was generated to compare the proposed solution with real IDS system. SVM algorithm was selected because of the recall rate 99.33% in detecting polymorphic NOOP's with low false alarm.

* Corresponding author: Tawfiq S. Barhoom
E-mail: tbarhoom@iugaza.edu

1.Introduction

Information Technology infrastructure suffering from various vulnerabilities threats especially zero-day (0day) vulnerabilities which is the main reason in destroying systems, leaks information, and causes financial destruction. Buffer overflow is the most famous type of vulnerability which can hijack systems, execute remote applications, and spreading worms. Figure 1 shows that buffer overflow appears in a high severity and dangerous vulnerability that is used in cyber-attacks [1] [2]. This type of attacks forced security companies and security researchers to find optimal solutions that can protect complete solution that can protect and avoid systems from being hacked by buffer overflow.

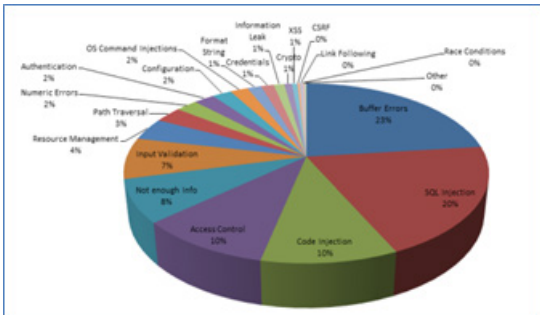

Figure 1 Top Vulnerability types with a high severity [2]

Buffer overflow is caused by bad programming practices used from programmers through working with memories without any boundary check, so while writing data to a buffer, it overruns the buffer's boundary and overwrites adjacent memory locations [3].

According to this issue, researchers started putting solutions by advising, using alternative programming languages that have built-in protection against accessing or overwriting data in any part of a memory. As C and C++ provide ability to work with the memory without checking the boundaries of buffers in writing, beside that, advise to stop using standard library functions and uses safe libraries that check boundaries [4], Microsoft provided application programming interface (API) routine to use Point Guard, implemented executable space protection in the core of operating systems, created data execution prevention (DEP), invented address space layout randomization (ASLR), Return Oriented Programming (ROP) prevent etc…. Inspite of these efforts, hackers always find ways, holes, and new techniques to skip these prevention techniques. To date, network intrusion detection systems detect and prevent such attacks by identifying worms and Shellcodes through using a fixed byte sequence of signature which is stored in updatable database of previously known worm's payload [5]. Concluding that there is no one solution for this threat, instead, we need dozens of solutions through which every solution solves one face from buffer overflow faces, so researchers use static analysis by analyzing the source code and dynamic analysis that analyses the applications on runtime. A point of view that handles this problem from another perspective by not working on the system itself but working on the network level and identify the packets transferred in the network that cause buffer overflow attacks. In this area there are lots of researches that detect and prevent the payloads on the network; but as usual there are techniques used by hackers to evade these approaches. Nowadays, there lots of engines that produce encrypted Shellcodes like those in Metasploit Framework [6], ecl-poly [7], AdMutate [8], or CLET [9]. By digging down into the structure of a Shellcode, there are main sections that must be in the Shellcode to make the overflow successful, which consist of NOP sled, payload, and return sled. Our work takes NOP sled section to identify the Shellcode while being transferred in the network, NOP section can consist of a huge probability of useless instructions which are generated and obfuscated by Shellcode engines.

In this Paper, Bernoulli Naïve Bayes, Decision Tree, and SVM data mining algorithms are used to be trained on special selected features that areextracted from very large amounts of polymorphic NOPs in Shellcodes. This allows the classifier to know the patterns which identify this section of a Shellcode. Therefore, the proposed solution can alarm that the system under buffer overflow is being attacked.

The rest of this paper is organized as follows: section two; related work, section three; methodology, section four; experimentation, section five; the results of experimentation and section six concludes the paper.

2.Related Work

Overflow detection and prevention problems have been studied since the mid-nineties. However, many recent researches have been published to solve this hot problem.

2.1.Static Analysis:

Zhao, Z. et al. proposed a technique for modeling Shellcode detection and attribution through the instruction of sequence abstraction, which extracts coarse-grained features from an instruction sequence. This technique uses Markov model for Shellcode detection and supports vector machines for encoded Shellcode attribution [10]. The solution is based on static analysis and supervises machine learning techniques, to extract coarse-grained features used instead of byte patterns. The evaluation shows that this solution can detect all types of un-encoded Shellcodes from their dataset and can attribute encoded Shellcodes to their origin engine with high accuracy. Despite the efforts that got our attention, they used a small sample for training and all of these samples were from only one engine that also uses all Shellcode sections in the training because the model works on known payloads and returns ranges. But it is bypassed by adding low NOOP's altogether with unknown payloads in the Shellcode, so it can spoof it and pass.

Gamayunov, D. et al. proposed Racewalk algorithm which is a significant modification of the Stride algorithm that had linear computational complexity [11] [12]. It claims novelty of NOOP-sled detection using IA-32 instruction frequency analysis and SVM-based classification.

This approach reduces the false positive, and the speed of operation is 1Gbps. The main idea in this algorithm is the NOOP-zone which consists of generally useless instructions that allow the return address zone to be in the correct stack segment; because this varies from system to system, so it detects the sled candidates and sends them to SVM-based instruction frequency analyzer. Using only Four Shellcode engine generators, this algorithm was applied.

Still, there are many defects like detecting NOOPs of IA-64 and not being able to detect the Shellcode construction methods that do not rely on NOOP-sleds or using self-modified sleds that are not supported and bypassed by spoofing classifiers in the same instruction set but with unusual operands.

2.2.Dynamic Analysis:

Fen, Y. presented a method that uses randomization based on data protection through protection of pointers and arrays, because of buffer overflow nature which depends on exceed writing on the limited area and to populate the return address, they use randomization on the arrays and pointers in program space to protect buffers, point data, and return address. This randomization is applied on the source by using XOR encryption for all the arrays and buffers. So, when the overflow happens, the target will be an encryption value which couldn't be point to, then the attack would fail. This approach is applied on the coding time to protect your self-application from being used in any type of buffer overflow attacks on the systems; but the major problem still exists; the applications from the shelf or on the operating system itself [13].

Khodaverdi, J. et al. proposed robust run time heuristic for detecting those Shellcodes which are hard-coded addresses; taking into consideration the fact that there are still too many users using older versions of windows that are not protected by Address Space Layout Randomization (ASLR) -enabled Windows. They used a custom emulator which supports the execution of IA-32 instructions, and they repeated the execution multiple times starting from each location of the input stream, to find all possible executable sequences of instructions in the input stream and detect any hard corded addresses that point to the stack pointer. Their evaluation results showed low false positive on 10 million random binaries [14].

They assumed using this emulator in a host level to detect the attacks, and for better performance. However, this approach could not detect return oriented programming (ROP).

2.3.Quantitative Analysis:

Song, Y. et al. presented a quantitative analysis of the strength and limitations of Shellcode polymorphism and described the impact of these techniques in context of learning-based IDS systems. They focused on two methods: Shellcode encryp-

tion-based and targeted blending attacks; because these two types are used in wild attacks and are successive in evade IDS sensors [15]. Their paper demonstrates metrics to measure the effectiveness of modern polymorphic engines and provide insights into their designs. The paper dived in the construction of many Shellcode types to understand the overall issue, and after that analyzed the polymorphic engines –six of them- and by generating 10000 unique samples they plotted visualization images for each engine outputs to extract the pattern they used in creating the op codes. Also, they combined two engines that use polymorphism and blended them into one engine that they called A Hybrid Engine. They simply used CLET to cipher the Shellcode, then hid CLET's decoder with ADMmutate and used ADMmutate's advanced NOP sled generator and showed how the attackers can blend between many engines to generate new patterns. After that present newed a design to detect the modern obfuscation techniques. This paper allows us to go throw the inside of designing the polymorphic Shellcode engines.

2.4.Hybrid Analysis:

Yuan, J. et al. proposed a method that uses static analysis (source code analysis) with the dynamic test (test a program while it is running), so this approach strikes a proper balance between static and dynamic analysis to identify buffer overflow vulnerabilities in a binary code (IA-32) without a source code [16].

They used two steps in their approach, first found some potential weakness locations then tested every potential weakness locations to reduce the false positive. After disassembly programs they went through many steps including identifying function call relations, analysing stack space, analyzing parameters, the use of local buffer, and finally determining the overflow function by using BugScam that can detect functions utilized in the binary file like Strcpy and so on.

And on the dynamic use Ollydbg to populate these functions that were identified before in static to see if it would check bounders or their overflow. Testing results shows low false alarm. We see that this approach can handle the stack overflow, and heap overflow can be a successful and needs us to put all the binaries of the organizations to this analysis to allow it to know if there is the ability to buffer overflow and this is not easily achieved!

The proposed solution is different than those solutions by depending on special feature extraction to make the classifier algorithm know the pattern of the polymorphic NOP generated.

3.Methodology

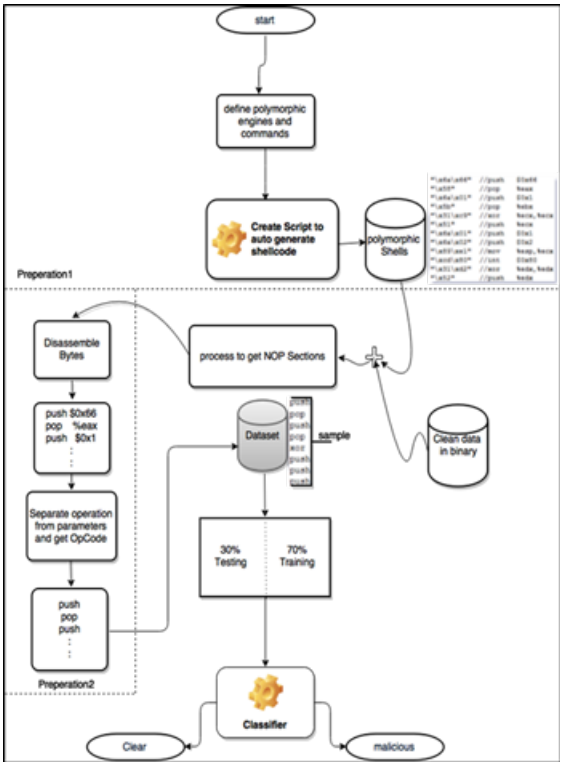

Figure 2 The Proposed Solution

The proposed solution depends on data mining classification techniques. It identifies malicious packets transferred in the network by using the first part from the three parts of Shellcode, which is known as NOP sled and specifically the polymorphic NOPs. This type of NOPs is applied as an advance fully undetectable attack.

Figure 2 shows the steps of the solution that are followed to achieve the target. Firstly, defining the polymorphic engines. Metasploit Shellcode engines (SINGLE-BYTE and OPTY2) are chosen and which have architecture IA-32. Then implementing a script that applies automatic generation on the engines with all possible parameters. This step produces significant amount of polymorphic Shellcodes that are generated and

labeled as malicious.

These Shellcode are CPU instructions in hexadecimal format, which do nothing other than forwarding the execution of payloads to the next instructions. In the same time, we collect massive files with different types of data to be the clean data and convert them to HEX.

After that moving to the next preparation by using Capstone Engine to disassemble all the hexadecimals of the two labels. This disassembly will convert the HEX to sequence of CPU instructions (assembly).

The last step here in building the dataset is to extract the features that aree used in the classification algorithms.

So, just extracting the operation code of all assembly instructions for the two labels. This leads to having a dataset that looks like Figure 3. Each line represent a file with its label.

({and, dec, jg, xor, sub, mov, jge, jl, jecxz, add, adc, lahf, xchg, jae, jno, loop, cmp}, **'clean'**)
({and, lea, dec, inc, sub, salc, mov, sbb, jecxz, add, test, adc, jg, das, xchg, xor, cwde, or, cmp}, **'clean'**)
({and, lea, jnp, inc, stc, jp, mov, cwde, jo, das, xchg, jg, dec, aad}, **'malicious'**)
({jns, and, xor, sub, stc, mov, js, clc, rcl, jbe, xchg, mul, jg, jno, inc}, **'malicious'**)

Figure 3 Four Samples of Dataset

By going forward, all these features are ordered without repetition as shown in Figure 4. This sequence is the header of the classification input matrix, listing the instructions like this without respect to the order and the length of the input; because real environments systems couldn't determine the length of Shellcode or from where it's starting.

aad,adc,add,and,clc,cmp,cwde,das,dec,inc,jae,jbe,jecxz,jg,jge,jl,jno,jnp,jns,jo,jp,js,lahf,lea,loop,mov,mul,or,rcl ,salc,sbb,stc,sub,test,xchg,xor

Figure 4 Feature names header

Then dataset refined to be suitable to the classification method by converting its records to Boolean matrix which is produced from Formula 1.

$$M_{n,j} = \begin{cases} 1, & n \text{ is available in } j \text{ record} \\ 0, & n \text{ is not available in } j \text{ record} \end{cases}$$

Where,

n is the index of feature in the features name header.

j is the index of feature records.

Formula 1 Record to Boolean Value Conversion

Representing dataset example to the Boolean by using Formula 1. Producing Boolean matrix as shown in Figure 5.

The matrix in Figure 5 consist of rows that are equal to the dataset files count that appear in Figure 3 and the columns is the number of the features in Figure 4. So by checking the availability of each feature in the record, we can identify the matrix element is 0 or 1.

$$\begin{bmatrix} 0&1&1&1&0&1&0&0&1&0&1&0&1&0&1&1&1&1&1&0&0&0&0&0&0&0&1&0&1&1&0&0&0&0&0&0&0&1&0&1&1 \\ 0&1&1&1&0&1&1&1&1&1&1&0&0&1&1&0&0&0&0&0&0&0&0&0&1&0&1&0&1&0&1&1&0&1&0&1&1&1&1 \\ 1&0&0&1&0&0&1&1&1&1&0&0&0&1&0&0&0&1&0&1&1&0&0&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0 \\ 0&0&0&1&1&0&0&0&0&1&0&1&0&1&0&0&1&0&1&0&1&0&0&1&0&0&0&1&1&0&1&0&0&1&1&0&1&1 \end{bmatrix}$$

Figure 5 Matrix of Boolean Weighing of Four Example Records

The last step in the solution is to pass this matrix to the classification algorithm. Classification methods are used such as SVM, Decision Tree, and Bernoulli NB to find which of them is the most method that suites the target of efficient malicious packets detection.

Representing how Decision Tree model will be applied on the four records of Boolean matrix shown in Figure 6.

This figure shows that the algorithm took a second feature as a root because if the record that has (adc) instruction, it will be clean and malicious if not available.

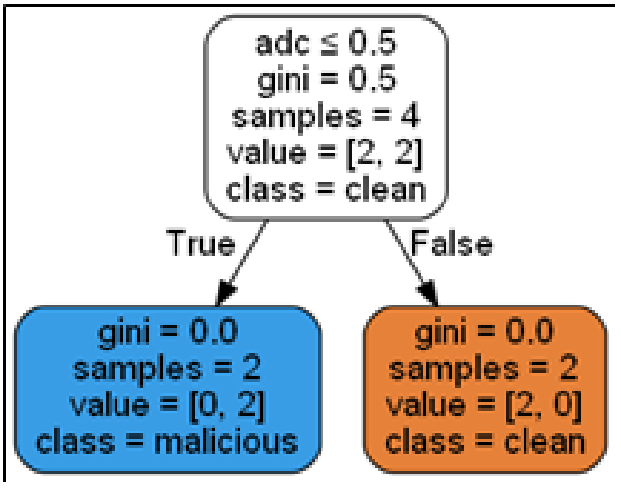

Figure 6 Output Representation of Decision Tree Applying on the four Samples.

The 4 records model is a small example that can be larger according to how large the dataset is. In Figure 7 the representation of a Decision Tree is applied on twenty-eight samples' matrix as another example.

Figure 7 Decision Tree Model for Twenty eight Samples

4.Experimentation

4.1.Corpus:

The corpus contains 500,000 malicious files and 72,000 clean files. Malicious corpus is generated from Metasploit polymorphic NOP's engine for 1 byte and multi bytes (OPTY2) with a max of 5000 bytes. Also collected clean files from various types.

4.2.Setup Tools:

Installed python 2.7 on 2.5 GHZ core I7 machine with 10Gb RAM. Using script tools available in python, we installed NLTK by this command (pip install nltk). After that installed scikit-learn tool by applying the command (pip install -U scikit-learn).

4.3.Preprocessing:

We collected malicious files as well as clean files which have the hexadecimal representation converted to assembly lines using Capstone Engine [17], then got the operation code of each line as it's the selected feature that we need to apply the experiments on it.

Malicious dataset has large number of files compared to the clean data so dataset shuffled and chose 70,000 records randomly.

Processing:

Script implemented to use the algorithms API of SKLEARN Library to process this dataset with respect to training the algorithm and testing it and calculating the accuracy, precision, and recall to evaluate each algorithm performance and determine its effectiveness. Precision is the percentage of predicted documents class that is correctly classified. Recall is the percentage of the total documents for the given class that are correctly classified. Also, computed the F-measure a combined metric that takes both precision and recall into consideration [18].

5.Experimentation Results

This section presents the results of three experiments using the three different algorithms which are: SVM, BNB, and DT. Algorithms training applied on 70% of the two labeled and used the rest (30%) of the dataset to measure the performance and efficiency of each algorithm.

Table 1 illustrates the performance measurement results of each algorithm according to the precision (TP/TP+FP), recall (TP/TP+FN), and F-measure (2*precision*recall / precision + recall) metrics. From this, results found that SVM has the highest rate 99.3% of correctly malicious prediction from all of real malicious and this computed from recall. Beside that, they found that SVM had 9.5% of false alarm.

The Accuracy (TP+TN/TP+TN+FN+FN) metrics are computed and listed in Table 2. It shows clearly that SVM has the highest accuracy with 94.91%.

Table 1 Precision, Recall, and F-measure of algorithms.

	Precision		Recall		F-measure	
	Pos	Neg	Pos	Neg	Pos	Neg
BNB	90.9%	96.78%	97%	90.33%	93.87%	93.44%
SVM	91.27%	99.26%	99.3%	90.5%	95.13%	94.68%
DT	91.93%	94.82%	95%	91.66%	93.44%	93.22%

Table 2 Accuracy and Execution Time

	accuracy	Execution Time	
		Training	Testing
BNB	93.66%	2.7039 sec	1.671 sec
SVM	94.91%	3.968 sec	1.734 sec
DT	93.33%	3.233 sec	1.405 sec

6.Conclusion

We demonstrated how dangerous the buffer overflow is, and how hackers can be employing the weapons of polymorphic Shellcodes to hack the systems and bypass security that can catch Shellcodes.

Data mining classification is used in this solution.

This solution depends on the idea of getting the op-code of the CPU Intel architecture instruction sets for the polymorphic sled NOOPs of 32-bit and applying the classification on it. Our solution depends on a self-generated dataset from Metasploit polymorphic NOOPs engines. Applying different classification algorithms on the dataset to get the perfect method that can deal with the problem.

Solution experiments illustrated high accuracy in detecting malicious data on the network with low false alarm for most of the algorithms we used. SVM was chosen as the best classification algorithm that can handle this issue because of it's 94% accuracy and getting 99.33% of recall metrics and the low false alarm we get.

Our solution shows significant results comparing against signature based on SNORT IDS which we compared against 1000 packets of polymorphic Shellcodes and the IDS classified 50.2% packets as harmful packet. On the other hand, our solution detects most of these packets with a close rate of 94%.

References

[1] National Institute Of Standards and Technology, "https://nvd.nist.gov/home.cfm," 2016. [Online].

[2] Y. Younan, "25 Years of Vulnerabilityies: 1988- 2012," Sourcefire Vulnerability Research Team (VRTTM), 2013.

[3] "Buffer Overflow In Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Buffer_overflow. [Accessed March 2016].

[4] Spafford, E. H. (1989). The Internet worm program: An analysis. ACM SIGCOMM Computer Communication Review, 19(1), 17-57.

[5] "SNORT," 2016. [Online]. Available: www.snort.org. [Accessed 2016].

[6]. "www.metasploit.org," [Online].

[7] Y. Gushin, "http://www.ecl-labs.org/papers/ecl-poly.txt," 2008. [Online].

[8] K2, "http://www.ktwo.ca/security.html," [Online].

[9] CLETteam, "Polymorphic Shellcode Engine Using Spectrum Analysis.," Phrack Magazine, 2003.

[10] Zhao, Z., & Ahn, G. J. (2013, October). Using instruction sequence abstraction for shellcode detection and attribution. In Communications and Network Security (CNS), 2013 IEEE Conference on (pp. 323-331). IEEE.

[11] Gamayunov, D., Quan, N. T. M., Sakharov, F., & Toroshchin, E. (2009, November). Racewalk: fast instruction frequency analysis and classification for shellcode detection in network flow. In Computer Network Defense (EC2ND), 2009 European Conference on (pp. 4-12). IEEE.

[12] Akritidis, P., Markatos, E., Polychronakis, M., & Anagnostakis, K. (2005). Stride: Polymorphic sled detection through instruction sequence analysis. Security and Privacy in the Age of Ubiquitous Computing, 375-391.

[13] Fen, Y., Fuchao, Y., Xiaobing, S., Xinchun, Y., & Bing, M. (2012). A new data randomization method to defend buffer overflow attacks. Physics Procedia, 24, 1757-1764.

[14] Khodaverdi, J., & Amin, F. (2013). A Robust Behavior Modeling for Detecting Hard-coded Address Contained Shellcodes. International Journal of Security & Its Applicetions, 7(5).

[15] Song, Y., Locasto, M. E., Stavrou, A., Keromytis, A. D., & Stolfo, S. J. (2010). On the infeasibility of modeling polymorphic shellcode. Machine learning, 81(2), 179-205.

[16] Yuan, J., & Ding, S. (2011, May). A method for detecting buffer overflow vulnerabilities. In Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on (pp. 188-192). IEEE.

[17] Capstone, "Capstone The Ultimate Disassembler," August 2010. [Online]. Available: http://www.capstone-engine.org. [Accessed 28 April 2016].

[18] Makhoul, J., Kubala, F., Schwartz, R., & Weischedel, R. (1999, February). Performance measures for information extraction. In Proceedings of DARPA broadcast news workshop (pp. 249-252).