Chapter 1

A DATA MINING TECHNIQUE TO DETECT REMOTE EXPLOITS

Mohammad M. Masud, Latifur Khan, Bhavani Thuraisingham, Xinran Wang, Peng Liu and Sencun Zhu

Abstract We design and implement *DExtor*, a *D*ata Mining based *Ex*ploit code detec*tor*, to protect network services. The main assumption of our work is that normal traffic into the network services contain only data, whereas exploit code contains code. Thus, the "exploit code detection" problem reduces to "code detection" problem. DExtor is an application-layer attack blocker, which is deployed between a web service and its corresponding firewall. The system is first trained with real training data containing both exploit code, and normal traffic. Training is performed by applying binary disassembly on the training data, extracting features, and training a classifier. Once trained, DExtor is deployed in the network to detect exploit code and protect the network service. We evaluate DExtor with a large collection of real exploit code and normal data. Our results show that DExtor can detect almost all exploit code with negligible false alarm rate. We also compare DExtor with other published works and prove its effectiveness.

Keywords: Exploit code, data mining, Classifier, server attack, software vulnerability

1. Introduction

Remote exploits are a popular means for attackers to gain control of hosts that run vulnerable services/software. Typically, a remote exploit is provided as an input to a remote vulnerable service to hijack the control-flow of machine-instruction execution. Sometimes the attackers inject executable code in the exploit that are executed after a successful hijacking attempt. We will refer to these code-carrying remote exploits as *exploit code*.

There are several approaches for analyzing network-flows to detect exploit code [1, 2, 8–11]. If an exploit can be detected and intercepted on its way to a server process, then an attack will be prevented. This approach is compatible with legacy code and does not require any change to the underlying computing infrastructure. Our solution, *DExtor*, follows this perspective. It is a data mining approach to the general problem of exploit code detection. The problem may be briefly described as follows.

Usually, an exploit code consists of three different parts: 1) a NOP sled at the beginning of the exploit, 2) a payload in the middle and 3) return addresses at the end. The NOP sled is a sequence of NOP instructions; the payload contains attacker's code; the return addresses point to the code. Thus, an exploit code always carries some valid executables in the NOP sled and payload, and considered as an *attack* input to the corresponding vulnerable service. Inputs to a service that do not exploit its vulnerability are considered as *normal* inputs. For example, with respect to a vulnerable HTTP server, all benign HTTP requests are 'normal' inputs, and requests that exploit its vulnerability are 'attack' inputs. If we assume that 'normal' inputs may contain only data, then the "exploit code detection" problem reduces to "code detection" problem. In order to justify this assumption, we refer to [2]. They maintain that "the nature of communication to and from network services is predominantly or exclusively data and not executable code." However, there are exploits that do not contain code; such as integer overflow exploits, or return-to-libc exploits. We do not deal with these kinds of exploits. It is also worth mentioning that code detection problem is fundamentally different from "malware detection" problem, which tries to identify the presence of malicious content in an executable.

We apply data mining to detect the presence of code in an input. We extract three kinds of features: *Useful Instruction Count*(UIC), *Instruction Usage Frequencies*(IUF), and *Code vs Data Length*(CDL). These features are explained in details in section 3.3. Data mining is applied in order to differentiate between the characteristics of 'attack' inputs from 'normal' inputs based on these features. The whole process consists of several steps. First, training data are collected that consist of real examples of 'attack' (e.g. exploits) and 'normal' (e.g. normal HTTP requests) inputs. The data collection process is explained in section 4.1. Second, all the training examples are disassembled, applying the technique explained in section 3.2. Third, features are extracted from the disassembled examples and a classifier is trained to obtain a classification model. A number classifiers are applied such as *support vector machine* (SVM), *Bayes net*, *decision tree* (J48), and *boosted J48*, and the best of them is chosen. Finally, DExtor is deployed in a real networking envi-

ronment. It intercepts all inputs destined to the network service that it protects, and tests them against the classification model to determine whether they are 'normal' or 'attack'.

The next obvious issue is how we deploy DExtor in a real networking environment and protect network services. DExtor is designed to operate at the application layer, and can be deployed between the server and its corresponding firewall. It is completely transparent to the service that it protects. Meaning, no modification at the server is required. It can be deployed as a standalone component, or coupled with a proxy server as a proxy filter. We have deployed DExtor in a real environment as a proxy, protecting a web server from attack. It successfully blocks 'attack' requests in real time. We evaluate our technique in two different ways. First, we apply a five-fold cross validation on the collected data, which contains 9,000 exploits and 12,000 normal inputs, and obtain a 99.96% classification accuracy and 0% false positive rate. Second, we test the efficacy of our method in detecting new kinds of exploits. This also achieves high detection accuracy.

Our contributions are as follows. First, we identify different sets of features, and justify their efficacy in distinguishing between 'normal' and 'attack' inputs. Second, we show how a data mining technique can be efficiently applied in exploit code detection. Finally, we design a system to protect network services from exploit code and implement it in a real environment. In summary, DExtor has several advantages over existing exploit-code detection techniques. First, DExtor is compatible with legacy code, and transparent to the service it protects. Second, it is readily deployable in any system. Although currently it is deployed on windows with Intel 32-bit architecture, it can be adapted to any operating system and hardware architecture only by modifying the disassembler. Third, DExtor do not require any signature generation/matching. Finally, DExtor is robust against most attack-side obfuscation techniques, as explained in section 5.

Our technique is readily applicable to digital forensics research. For example, after a server crash, we may use our technique to analyze the network traffic that went to the server before the crash. Thus, we may be able to determine whether the crash was caused by any code-carrying exploit attack. We may also be able to determine the source of the attack.

The rest of the paper is organized as follows: section 2 discusses related work, section 3 explains the proposed method in detail, section 4 discusses the dataset and experimental results, section 5 discusses security and implementation issues, and section 6 summarizes our work with future research direction.

2. Related Work

There are many techniques available for detecting exploits in network traffic and protecting network services. Three main categories in this direction are signature matching, anomaly detection, and machine-code analysis.

Signature matching techniques are the most prevailing and popular. Intrusion Detection Systems (IDSs) Snort [8] and Bro [1] follow this approach. They maintain a signature-database of known exploits. If any traffic matches a signature in the database, the IDS raises an alert. These systems are relatively easy to implement, but they can be defeated by new exploits, as well as polymorphism and metamorphism. On the contrary, DExtor do not depend on signature matching.

Anomaly detection techniques detect anomalies in the traffic pattern and raises alert when an anomaly is detected. Wang et al. [11] propose a payload based anomaly detection system called PAYL, which first trains itself with normal network traffic and detects exploit code by computing several byte-level statistical measures. Some other anomaly-based detection techniques are the improved version of PAYL [10], and FLIPS [5]. DExtor is different from anomaly-based intrusion detection systems for two reasons. First, anomaly-based systems train themselves using the 'normal' traffic characteristics, and detect anomalies based on this characteristic. On the other hand, our method considers both 'normal' and 'attack' traffic to build a classification model. Second, we consider instruction patterns, rather than raw byte patterns, for building a model.

Machine-code analysis techniques apply binary disassembly and static analysis on network traffic to detect presence of executables. DExtor falls in this category. Toth and Kruegel [9] use binary disassembly to find long sequences of executable instructions and identify presence of a NOP sled. DExtor also applies binary disassembly, but it does not need to identify NOP sled. Chinchani et al. [2] detect exploit code based on the same assumption as DExtor that normal traffic should contain no code. They apply disassembly and static analysis, and identify several structural patterns and characteristics of code-carrying traffic. Their detection approach is rule-based. On the other hand, DExtor do not require generating or following rules. SigFree [12] also disassembles inputs to server processes and applies static analysis to detect presence of code. SigFree applies *code abstraction* technique to detect useful instructions in the disassembled byte-stream, and raises an alert if the useful instruction count exceeds a predetermined threshold. DExtor applies the same disassembly technique as SigFree. But DExtor does not detect presence of code based on a fixed threshold. Rather, it applies data mining to

extract several features and learn to distinguish between normal traffic and exploits based on these features.

3. Description of the Proposed Method

This Section describes DExtor architecture, feature extraction and classification processes.

3.1 DExtor Architecture

The architecture of DExtor is illustrated in figure 1. DExtor is deployed in a network between the network service and its corresponding gateway/firewall. It is first trained offline with real instances of attack (e.g. exploits) and normal (e.g. normal HTTP requests) inputs, and a classification model is obtained. Training consists of three steps: disassembly, feature extraction, and training with a classifier. After training, DExtor is deployed in the network and all incoming inputs to the service are intercepted and analyzed online. Analysis consists of three steps: disassembly, feature extraction and testing against the model. These processes are explained in details in this section.

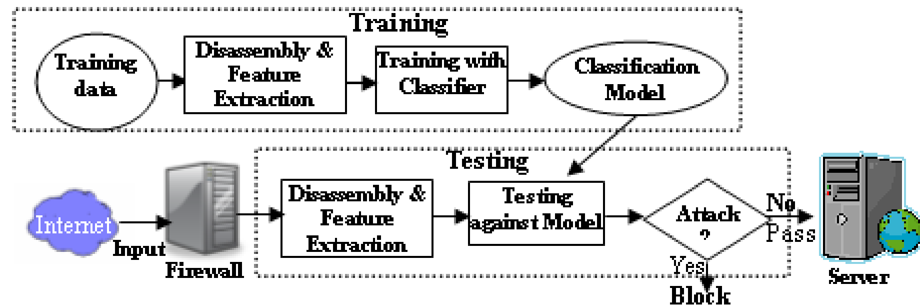

Figure 1. DExtor architecture

3.2 Disassembly

The disassembly algorithm is similar to [12]. Each input to the server is considered as a byte sequence. There may be more than one valid assembly instruction sequences corresponding to the given byte sequence. The disassembler applies a technique called the "instruction sequence distiller" to filter out all redundant and illegal instruction sequences. The main steps of this process are as follows: *Step 1. Generating instruction sequences*, *Step 2. Pruning subsequences*, *Step 3. Discarding smaller sequences*, *Step 4. Removing illegal sequences*, and *Step 5. Iden-*

tifying useful instructions. Readers are requested to consult [12] for details.

3.3 Feature Extraction

Feature extraction is the heart of our data mining process. We have identified three important features based on our observation and domain-specific knowledge. These are: *Useful Instruction Count* (IUC), *Instruction Usage Frequencies* (IUF), and *Code vs Data Lengths* (CDL). These features are described here in details.

Useful Instruction Count (UIC) This is the number of useful instructions found in step 5 of the disassembly process. This number is important because a real executable should have higher number of useful instructions, whereas data should have less or zero useful instructions.

Instruction Usage Frequencies (IUF) In order to extract this feature, we just count the frequency of each instruction that appears in an example (normal or attack). Intuitively normal data should not have any bias/preference toward any specific instruction or set of instructions. Thus, the expected distribution of instruction usage frequency in normal data should be random. On the other hand, an exploit code is supposed to perform some malicious activities in the victim machine. So, it must have some bias/preference toward a specific subset of instructions. Thus, the expected distribution of instruction usage frequencies should follow some pattern. This idea is also supported by our observation of the training data, which is illustrated in section 1.4.4.

Code vs Data Length (CDL) As explained earlier, an exploit code has three different regions: the NOP sled, the payload, and the return addresses. Following from this knowledge and our observation of the exploit code, we divide each input instance into three regions or 'zones': *bzone* or the beginning zone, *czone* or the code zone, and *rzone* or the remainder zone. 'bzone' corresponds to the first few bytes in the input that could not be disassembled, and probably contains only data. For example, the first 20 bytes of the exploit in figure 2. 'czone' corresponds to the bytes after 'bzone' that were successfully disassembled by the disassembler, and probably contains some code (e.g.: bytes 20-79 in figure 2). 'rzone' corresponds to the remaining bytes in the input after 'czone' that could not be disassembled, and probably contains only data (e.g.: last 20 bytes in figure 2). We observe that the normalized lengths (in bytes) of these three zones follow a certain distribution for

'attack' inputs, which is different from that of the 'normal' inputs. These distributions are also illustrated in section 4.4.

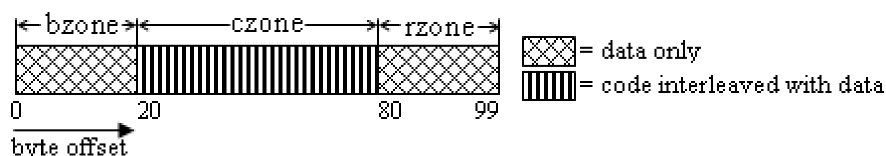

Figure 2. Three zones of an input instance

Intuitively, normal inputs should contain code zone at any location with equal probability. Meaning, the expected distribution of 'bzone' and 'rzone' should be random in normal inputs. Also, normal inputs should have few or no code. Thus, 'czone' length should be near zero. On the other hand, exploit code are restricted to follow a certain pattern for the code zone. For example, the exploit code should begin with the NOP sled, necessitating the 'bzone' length to be equal to 0. Also, 'czone' length for exploit codes should be higher than normal inputs. In summary, the patterns of these three zones should be distinguishable in normal and attack inputs.

3.4 Combining features and Compute Combined Feature Vector

The feature vectors/values that we have computed for each input sample are: I) UIC a single integer, II) IUF - containing K integer numbers representing the frequencies of each instruction, where K is the total number of different instructions found in the training data. III) CDL features, containing 3 real values. So, we have a collection of $K + 4$ features, of which the first $K + 1$ feature-values are integer, and the last 3 are real. These $K + 4$ features constitute our combined feature vector for an input instance.

3.5 Classification

We use Support Vector Machine (SVM), Bayes Net, decision tree (J48) and Boosting for the classification task. These classifiers are found to have better performances in our previous works related to malware detection. The rationale for using these classifiers is obvious. First, SVM is more robust to noise and high dimensionality. Besides, it can be fine-tuned to perform efficiently on a specific domain. Decision tree has a very good feature-selection capability. Besides, it is much faster than

many other classifiers, both in training and testing time. Bayes Net is capable of finding the inter-dependencies between different attributes. Boosting is particularly useful because of its ensemble methods.

4. Experiments

This section describes data collection, implementation, and evaluation of the proposed method.

4.1 Data Set

The data set contains real exploit code as well as normal inputs to web servers. We obtain the exploit codes as follows. First, we generate 20 un-encrypted exploits using the Metasploit framework [7]. Second, we apply nine polymorphic engines "ADMmutate" [6], "clet" [3], "Alpha2", "CountDown", "JumpCallAdditive", "Jumpiscodes", "Pex", "PexFnstenvMov", "PexFnstenvSub" on the un-encrypted exploits. Each polymorphic engine is applied to generate 1000 exploits. Thus, we obtain a collection of 9,000 exploit code. We collect the normal inputs from real traces of HTTP request/responses to/from a web server. In order to collect these traces, we install a client-side proxy that can monitor and collect all incoming and outgoing messages. Thus, the normal inputs consist of a collection of about 12,000 messages containing HTTP requests / responses. HTTP responses consist of texts (.javascript, .html, .xml), applications (x-javascript, pdf, xml), images (.gif, .jpeg, .png), sounds (.wav), and flash. Thus, we try to make the dataset as diverse, realistic, and unbiased as possible to get the flavor of a real environment.

We perform two different kinds of evaluation on the data. First, we apply a 5-fold cross validation. and obtain the accuracy, false positive, and negative rates. Second, we test the performance of the classifiers on new kinds of exploits. This is done as follows: a classifier is trained using the exploits obtained from eight engines, and tested on the exploits from the ninth engine. This is done nine times by rotating the engine in the test set. Normal examples were distributed in the training and test set with equal proportions. We report the performances of each classifier for all the nine tests.

4.2 Experimental Setup

We run our experiment with a 2.0GHz Machine with 1GB RAM on a Windows XP machine. Our algorithms are implemented in java, and compiled with jdk version 1.5.0_06. We use Weka [13] ML toolbox for the classification tasks. For SVM, we apply the C-Support Vector classifier (C-SVC) with a polynomial kernel and $gamma = 0.01$. For Bayes Net,

Table 1. Comparing performances among different features and classifiers

Feature	IUC	IUF	CDL	Comb
Metric	ACC/FP/FN	ACC/FP/FN	ACC/FP/FN	ACC/FP/FN
SVM	75.0/3.3/53.9	99.7/0.2/0.1	92.7/12.4/0.6	99.8/0.1/0.2
Bayes Net	89.8/7.9/13.4	99.6/0.4/0.4	99.6/0.2/0.6	99.6/0.1/0.9
J48	89.8/7.9/13.4	99.5/0.3/0.2	99.7/0.3/0.3	99.9/0.2/0.1
Boosted J48	89.7/7.8/13.7	99.8/0.1/0.1	99.7/0.3/0.5	99.96/0.0/0.1
SigFree	38.5/0.2/88.5			

we use a simple estimator with $alpha = 0.5$ and a hill-climbing search for the network learning. For J48, we use tree pruning with $C = 0.25$. Finally, we run 10 iterations of the AdaBoost algorithm to generate 10 models. We also evaluate the performances of each of the three features alone on each of these classifiers. To evaluate the performance of a feature on a classifier, we train and test the classifier with only that feature.

4.3 Results

We apply three different metrics to evaluate the performance of our method: *Accuracy* (ACC), *False Positive* (FP) and *False Negative* (FN), where ACC is the percentage of correctly classified instances, FP is the percentage of negative instances incorrectly classified as positive, and FN is the percentage of positive instances incorrectly classified as negative.

Table 1 shows a comparison among different features of DExtor. We see that accuracy of DExtor's Combined (shown under column *Comb*) feature classified with Boosted J48 is the best, which is 99.96%. Individual features have accuracies less than the combined feature for all classification techniques. Also, the combined feature has the lowest false positive, which is 0.0%, obtained from Boosted J48. The lowest false negative also comes from the combined feature, which is only 0.1%. In summary, the combined feature with Boosted J48 classifier has achieved near perfect detection accuracy. The last row shows the accuracy and false alarm rates of SigFree on the same data set. SigFree actually uses UIC with a fixed threshold (15). It is evident that SigFree has a low false positive rate (0.2%), but high false negative rate (88.5%), causing the overall accuracy to drop below 39%. Figure 3 shows the Receiver Operating Characteristic (ROC) curves of different features for BoostedJ48 classifier. ROC curves for other classifiers have similar characteristics, and are not shown due to space limitation. The area under the curve (AUC) is the highest for the combined feature, which is 0.999.

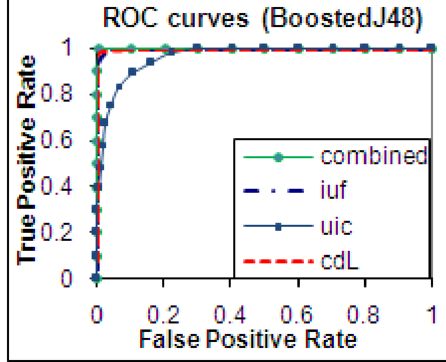

Figure 3. ROC curves of different features for BoostedJ48

Table 2. Effectiveness in detecting new kinds of exploits

Classifier	SVM	BNet	J48	BJ48
Metric	ACC/FP/FN	ACC/FP/FN	ACC/FP/FN	ACC/FP/FN
Admutate	86.4/0.2/31.7	57.4/0.0/100	98.2/0.0/4.3	99.7/0.0/0.6
Alpha2	99.9/0.07/ 0.0	56.4/0.0/100	56.4/0.0/100	56.4/0.0/100
Clet	100/0.0/0.0	99.6/0.07/0.8	99.9/0.1/0.0	99.9/0.07/0.0
CountDown	99.8/0.4/0.0	100/0.0/0.0	100/0.0/0.0	99.8/0.3/0.0
JmpCallAdditive	100/0.0/0.0	98.1/0.0/4.6	99.9/0.1/0.0	100/0.0/0.0
JumpisCode	99.4/0.08/1.4	96.2/0.08/8.8	99.9/0.07/0.0	99.9/0.07/0.1
Pex	99.7/0.2/0.4	99.4/0.0/1.4	99.8/0.2/0.2	99.8/0.1/0.3
PexFnStenvMov	99.9/0.0/0.0	99.1/0.0/2.1	99.9/0.07/0.1	99.9/0.0/0.2
PexFnStenvSub	99.7/0.2/0.3	99.3/0.0/1.7	99.8/0.08/0.1	99.9/0.08/0.0

Table 2 reports the effectiveness of our approach in detecting new kinds of exploits. Each row reports the detection accuracies and false alarm rates of one particular engine-generated exploits. For example, the row headed by ‘Admutate’ shows the detection accuracy (and false alarm rates) of exploits generated by the Admutate engine. In this case, the classifiers have been trained with the exploits from other eight engines. In each case, the training set contains 8,000 exploits and about 10,500 randomly selected normal samples, and the test set contains 1,000 exploits and about 1,500 randomly chosen normal samples. The columns headed by SVM, BNet, J48, and BJ48 show the accuracies (or false positive/false negative rates) of SVM, Bayes Net, J48, and Boosted J48 classifiers, respectively. It is evident from the table that all the classifiers could successfully detect most of the new exploits with 99% or better accuracy.

Running time: The total training time for the whole dataset is less than 30 minutes. This includes disassembly time, feature extraction time

and classifier training time. This amounts to about 37ms / KB of input. The average testing time / KB of input is 23ms for the combined feature set. This includes the disassembly time, feature value computation time and classifier prediction time. SigFree, on the other hand, requires 18.5ms to test per KB of input. Considering that training can be done offline, this amounts to only 24% increase in running time compared to SigFree. So, the price-performance trade-off is in favor of DExtor.

4.4 Analysis

As explained earlier, IUF feature observes different frequency distribution for the 'normal' and 'attack' inputs. This is illustrated in the leftmost chart of figure 4. This graph shows the 30 most frequently used instructions (for both kinds of inputs). It is seen that most of the instructions in this chart are more frequently used by the 'attack' inputs than 'normal' inputs. First five of the instructions have high frequencies (> 11) in 'attack' inputs, whereas they have near zero frequencies in 'normal' input. Next sixteen instructions in 'attack' inputs have frequencies close to two, whereas 'normal' inputs have near zero frequencies for these instructions. In order to mimic 'normal' input, an attacker should avoid using all these instructions. It may be very hard for an attacker to get around more than twenty most frequently used instructions in exploits and craft his code accordingly.

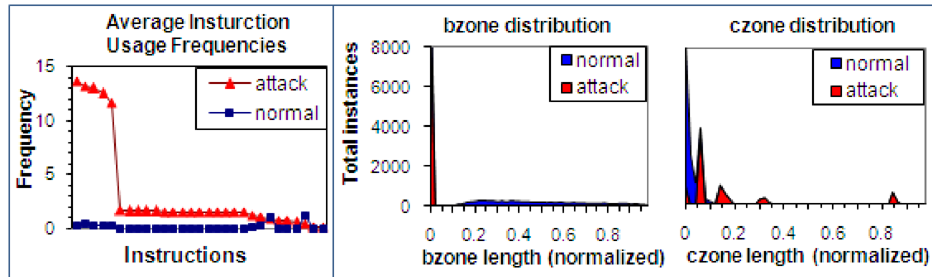

Figure 4. Left: average instruction usage frequencies (IUF) of some instructions. Right: distributions of 'bzone' and 'czone'

Similarly, we observe specific patterns in the distribution of the CDL feature-values. The patterns for 'bzone' and 'czone' are illustrated in the right two charts of figure 4. These are histograms showing the number of input samples having a particular length (as fraction of total input size) of 'bzone' or 'czone'. These histograms are generated by dividing the whole range ([0,1]) of 'bzone' (or 'czone') sizes into 50 equal-sized bins, and counting the total number of inputs instances that fall within

the range of a particular bin. By closely observing the histogram for *bzone*, we see that most of the 'attack' samples have *bzone* values in the first bin (i.e., [0,0.02)), whereas that of the 'normal' samples are spread over the whole range of values starting from 0.1. This means, if the attacker wants to mimic normal traffic, he should leave the first 10% of an exploit without any code. This may increase his chances of failure since the exploit should naturally start with a NOP sled. Again by closely observing the histogram for *czone*, we see that most of the 'normal' samples have 'czone' values within the range [0,0.05], whereas 'attack' samples mostly have 'czone' values greater than 0.05. Meaning, if the attacker wants to mimic normal traffic, he should keep his code length within 5% of the exploits length. For a 200-byte exploit, this would allot only 10 bytes for code - including the NOP sled. Thus, the attacker would have a hard time to figure out how to craft his exploit.

5. Discussion

In this section we discuss different security issues and the robustness/limitations of our system.

5.1 Robustness against obfuscations

Our technique is robust against 'Instruction re-ordering' since we do not care about the order of instructions. It is also robust against 'junk-instruction insertion' since it increases the frequency of instructions in the exploit. It is robust against instruction replacement as long as all the 'most frequently used' instructions are not replaced (as explained in section 1.4.4) by other instructions. It is also robust against register-renaming and memory re-ordering, since we do not consider register or memory locations. Junk byte insertion obfuscation is targeted to the disassembler, where junk bytes are inserted at locations that are not reachable at run-time. Our disassembly algorithm applies recursive traversal, which is robust to this obfuscation [4].

5.2 Limitations

DExtor is partially affected by the 'branch function' obfuscation. The main goal of this obfuscation is to obscure the control-flow in an executable, so that disassembly cannot proceed. Currently, there is no general solution to this problem. In our case, DExtor is likely to produce fragmented 'code blocks', missing some of the original code. This will not affect DExtor as long as the 'missed' block contains significant number of instructions.

Another limitation of DExtor is its processing speed. We evaluated the throughput of DExtor in a real environment, which amounts to 42KB/sec. This might seem unrealistic for an intrusion detection system that has to encounter Gigabits of data per second. Fortunately, we intend to protect just one network service, which is likely to process inputs much slower than this rate. We suggest two solutions to get around this limitation. First, using faster hardware and optimizing all software components (disassembler, feature extraction, classifier). Second, carefully excluding some incoming traffic from analysis. For example, any bulk input to the server having size greater than a few hundred KB is too unlikely to be an exploit code. Because, the length of a typical exploit code is within a few KB only. By applying both the solutions, DExtor should be able to operate in a real time environment.

6. Conclusion

In this paper we present DExtor, a data mining approach for detecting exploit code. We introduce three different kinds of features, and show how to extract them. We evaluate the performance of DExtor on real data and establish its efficacy in detecting new kinds of exploits. Our technique can also be applied to digital forensics research. For example, by analyzing network traffic, we may investigate whether the cause of a server crash was an exploit attack. However, there are several issues related to our technique that are worth mentioning.

First, a popular criticism against data mining is that it is heavily dependent on the training data supplied to it. So, it is possible that it performs poorly on some data, and shows excellent performance on other set of data. Thus, it may not be a good solution for exploit code detection, since there is no guarantee that it may catch all exploit codes with 100 percent accuracy. However, what appears to be the greatest weakness of data mining is also the source of a great power. If the data mining method can be fed with sufficient realistic training data, it is likely to exhibit near-perfect efficiency in classification. Our results justify this fact too. It is one of our future goals to continuously collect real data from network and feed them into the classification system. Since training is performed 'offline', longer training time is not a problem.

Second, we would like to relax our main assumption that "normal traffic carries only data". We propose adding a 'malware detector' to our model as follows. We would detect presence of code inside the traffic using our current model. If the traffic contains no code, then it is passed to the server. Otherwise, it is sent to the malware detector for a 'secondary inspection'. We have already implemented such detector

in one of our previous works. A malware detector detects malicious components inside an executable. If the malware detector outputs a green signal (i.e., benign executable), then we pass the executable to the server. Otherwise, we block and discard/quarantine the code.

Acknowledgement: The work was supported by AFOSR under contract FA9550-06-1-0045 and NSF CAREER CNS-0643906.

References

[1] Bro intrusion detection system. *http://bro-ids.org/*, Last accessed Aug 2007.
[2] R. Chinchani and E. V. D. Berg, a fast static analysis approach to detect exploit code inside network flows, *Proceedings of Recent Advances In Intrusion Detection*, 2005.
[3] T. Detristan, T. Ulenspiegel and Y. Malcom, Underduk, Polymorphic shellcode engine using spectrum analysis. *http://www.phrack.org/show.php?p=61&a=9*, Last accessed Jun 2007.
[4] C. Kruegel, W. Robertson, F. Valeur and G. Vigna, Static disassembly of obfuscated binaries, *Proceedings of USENIX Security* 2004.
[5] M.E. Locasto, K. Wang, A. D. Keromytis and S. J. Stolfo, Flips: Hybrid adaptive intrusion prevention. *Proceedings of Recent Advances In Intrusion Detection (RAID)*, 2005.
[6] S. Macaulay, Admmutate: Polymorphic shellcode engine, *http://www.ktwo.ca/security.html*, Last accessed July 2007.
[7] The Metasploit project. *http://www.metasploit.com*, Last accessed Sep 2007.
[8] Snort. *http://www.snort.org*, Last accessed Aug 2007.
[9] T. Toth and C. Kruegel, Accurate buffer overflow detection via abstract payload execution, *Proceedings of Recent Advances in Intrusion Detection (RAID)*, 2002.
[10] K. Wang, G. Cretu and S. J. Stolfo, Anomalous payload-based network intrusion detection and signature generation, *Proceedings of Recent Advances In Intrusion Detection (RAID)*, 2005.
[11] K. Wang and S. J. Stolfo, Anomalous payload-based network intrusion detection, *Proceedings of Recent Advances In Intrusion Detection (RAID)*, 2004.
[12] X. Wang, C. Pan, P. Liu and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. *USENIX Security*, 2006.
[13] Weka: Data mining software in java. *http://www.cs.waikato.ac.nz/ml/weka/*. Last accessed Sep 2007.