Countering Entropy Measure Attacks on Packed Software Detection

Xabier Ugarte-Pedrero, Igor Santos, Borja Sanz, Carlos Laorden and Pablo Garcia Bringas

S[3]Lab, DeustoTech - Computing
University of Deusto
Bilbao, Spain
Email: {xabier.ugarte, isantos, borja.sanz, claorden, pablo.garcia.bringas}@deusto.es

***Abstract*—Malware writers usually employ several techniques to evade detection. For the last years, the number of variants detected each day has increased significantly. Traditional approaches such as signature scanning, one of the most common techniques employed by anti-virus companies, are becoming inefficient for the high amount of samples found in the wild. In order to bypass this kind of filters, malware writers usually obfuscate and transform the code of their creations. One of the methods employed is executable packing, which consists in compressing or ciphering the real malicious code, and injecting a decryption routine into the executable that will load and decompress it at run-time. Entropy is a common heuristic for the detection of packed executables. High entropy values indicate a random distribution of the bytes that compose the executable, a property very common in compressed and ciphered data. Unfortunately, this entropy measure can be altered by different techniques that modify randomness. In this paper, we detail various attacks found on real Zeus family samples, one of the most powerful and spread malware families at this moment, which are protected by custom made packers. In addition, we describe a method for obtaining an alternative entropy measure more resilient to these techniques, and evaluate it for the classification of packed/not-packed executables, obtaining satisfactory detection and false positive rates.**

I. Introduction

Malicious software (or malware) is explicitly designed to harm computers. In the past, malware authors pursued fame and self-pride: a unique malware sample infected thousands or millions of computers. As opposite, today, thousands of malware samples are released everyday, and each of them infects few computers. According to PandaLabs[1], 73,000 new malware samples where found each day during the first quarter of 2011. The reason behind this fact is that malware authors' intentions have changed. For the last years, money is their main motivation. There are complex and well-organised networks of criminals that employ malicious software to obtain profits illicitly. The success of this new malware depends on its ability to bypass anti-virus tools and to stay undetected for enough time.

One common technique to bypass anti-virus solutions is packing. Packed executables store their malicious code as ciphered or compressed data with the aim of hiding it and evading signature scanning. The executables contain routines that load the original code at run-time and then execute it. A report elaborated by McAfee[2] claims that up to an 80 % of the malware analysed is packed.

Traditional techniques, such as signature scanning have also been applied to the detection of packed executables. Searching for certain byte sequences can be specially effective for well known tools such as UPX. PEiD[3] is an application used extensively, which is able to detect a wide range of packers. As well, Faster Universal Unpacker (FUU) [1] identifies the packer and then applies specific unpacking routines.

Unfortunately, the use of signatures, as for malware detection, are not effective with unknown or custom made packers. In fact, some malware families such as Zeus use multiple-layer packing techniques: the first layer of protection is performed by a custom packer, and a second layer is provided by a well-known packer [2]. What is more, according to Morgenstern and Pilz [3], the 35 % of malware is packed by a custom packer.

Static unpacking approaches analyse the executable without executing it. This technique is more efficient, but the quantity of data it can gather is limited due to the difficulty of some problems involved (e.g., machine code disassembly [4]). In contrast, dynamic unpacking approaches execute binaries in isolated environments (e.g. virtual machines or emulators) [5], to obtain the execution trace and observe their behaviour.

One typical technique commonly used by dynamic unpackers (e.g., Universal PE Unpacker [6] and OllyBonE [7]) is using heuristics. Nevertheless, since all the packers work in very different manners, these heuristics are not applicable to all of them. As opposite, other dynamic approaches are not so highly heuristic-dependent (e.g., PolyUnpack [8], Renovo [9], OmniUnpack [10] and Eureka [11]). Notwithstanding, these methods can be resource-consuming, and present limitations such as conditional execution of unpacking routines, a technique used for anti-debugging defense [12], [13], [14].

Besides, other approaches apply static techniques to detect whether a file is packed or not. PE-Probe [15], classifies files

[1]PandaLabs Quarterly Reports: Q1 2011. Available online: http://pandalabs.pandasecurity.com/pandalabs-quarterly-report-q1-2011/

[2]McAfee Whitepaper: The Good, the Bad, and the Unknown. 2011. Available online: http://www.mcafee.com/us/resources/white-papers/wp-good-bad-unknown.pdf

[3]PEiD. Available online: http://www.peid.info/

into packed and not-packed executables to extract different features for malware detection in each case. Perdisci et al. [16] proposed a method for the classification of packed executables based on heuristics commonly used by malware analysts, such as the number of standard sections, section permissions or entropy, as a previous step to the actual unpacking process. Similarly, we previously proposed several classification models for packed executable filtering based on anomaly detection and semi-supervised machine-learning techniques [17], [18], [19]. Compressed or ciphered data presents a higher randomness. For this reason, file entropy is a heuristic very extended and it constitutes one of the first measures checked in malware analysis to determine if an executable is packed [20]. As an alternative, Sun proposed a novel method for randomness analysis, generating randomness profiles to identify the packers employed to protect files [21].

In consideration of this background, we describe some of the attacks found on malware samples from the Zeus family, one of the most spread malware families at this moment [2], and propose a new method for detecting packed executable files. To this end, we apply an static method based on entropy analysis and byte histograms to generate alternative randomness profiles that allow an automatic analyser to ignore some of the attacks described. This method improves significantly the detection rates achieved by a simple file entropy analysis.

Summarising, our main contributions are: (i) we thoroughly describe the attacks to entropy analysis found in malware samples from the Zeus family, (ii) we propose a new method for measuring executable randomness that combines entropy analysis and byte histograms, (iii) we provide a method for measuring a randomness value for each executable and establish empirically a threshold to classify samples in 2 groups (packed and not-packed) by means of genetic algorithms, and finally, (iv) we measure the ability of our method to classify executable samples.

The remainder of this paper is organised as follows. Section II describes the attacks found on some Zeus family malware samples. Section III details the method designed. Section IV describes the experiments and presents results. Section V discusses the obtained results and their implications, and outlines the avenues for future work.

II. Attack Description

Zeus or ZBot is one of the most notorious and wide-spread trojans nowadays [2]. The anti-malware industry has a deep experience identifying and detecting it, but new samples are discovered everyday by analysts. In order to evade detection, these tools employ different techniques to avoid the traditional tests performed by anti-malware solutions. In our study, we found samples that implement different attacks to trick entropy analysis. Entropy is one of the most common checks malware analysts do to decide whether some code is packed or not. For a random variable X with n outcomes, $\{x_i : i = 1, \ldots, n\}$ the Shannon entropy $H(X)$ is calculated as $H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i)$ where $p(x_i)$ is the probability mass function of outcome x_i and b is the number of different symbols of the "ideal alphabet" used to measure source alphabets. The source alphabet used for measuring entropy is the set of 256 possible values that can be represented in a byte. According to information theory, 2 symbols are necessary and sufficient to encode data. Therefore, the entropy of the source alphabet, which represents the maximum possible randomness, is the number of symbols in the "ideal alphabet" needed to encode a symbol in the source alphabet: 8.

Many tools, (e.g., PEiD[4]), measure the entropy of the whole file and the sections of the executable in a coarse-grained style. One simple way to evade this detection technique is to append repeated bytes at the end of each section to modify the byte distribution and, thus, make the entropy lower.

Real malware samples employ even more sophisticated attacks. The techniques detailed below were found on Zeus malware family samples by reverse engineering and visual observation of byte histograms and randomness profiles in our self-developed entropy analysis tool.

1) **Random byte insertion**
This technique consists in the strategic insertion of bytes in the executable file. The bytes inserted are randomly selected from a reduced set of bytes, in such a way that a considerably high number of the bytes in the file pertain to that group. In this way, although some parts of the executable may be compressed or cyphered, if we measure the entropy of the file in a coarse-grained style (file entropy or section entropy), the computed value will be the typical of not-packed executables. More concretely, one of the samples found in our research had the half of its bytes (odd positions) artificially set to a value in the group {00, 20, 40, 60, 80, A0, C0, E0}. In Figure 1 we can observe the bytes inserted: the histogram shows that these bytes, represented as light bars, are much more frequent than the rest.
2) **Reduced source alphabet**
The second technique found was aimed at reducing the maximum possible entropy for the file either for coarse-grained and fine-grained entropy analysis. This technique consists in using only a subset of the symbols in the source alphabet, in such a way that the number of symbols necessary to represent the same information is higher. Let $\mathcal{S}$ be the source alphabet with $N = |\mathcal{S}|$ different possible symbols and $\mathcal{R}$ be a subset of the symbols in $\mathcal{S}$ with $M = |\mathcal{R}|$ different possible symbols, such that $N > M$. If we represent some information encoded with $\mathcal{S}$ with the set of symbols $\mathcal{R}$, we need $\log_M N$ symbols to represent the same information. As $N > M \Rightarrow \log_M N > 1$, the number of symbols needed to represent each symbol in the source alphabet is higher (> 1). As we need a higher number of symbols to represent the same information but we still use the source alphabet, the number of symbols in the "ideal alphabet" needed to represent the original information is higher. Therefore, if the entropy is computed for the

[4]PEiD. Available online: http://www.peid.info/

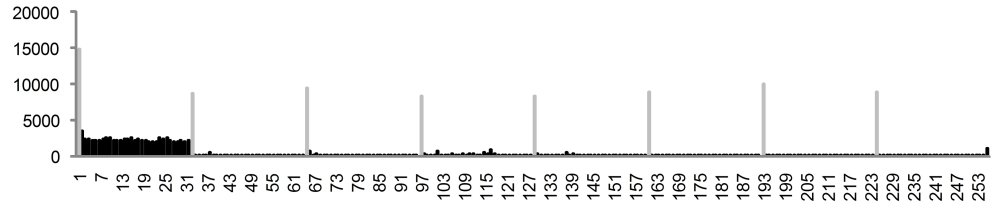

Fig. 1. Byte histogram of the analysed sample. The light bars represent the bytes inserted in the attack 1, while the dark bars from 0 to 31 represent the reduced set of bytes used for encoding the data, according to attack 2.

Fig. 2. Entropy profile of the analysed sample, that represents the randomness of the file in a fine-grained style.

complete source alphabet, the resulting value will be inferior (a more typical value in not-packed executables). In the histogram shown in Figure 1 we can see the dark bars (bytes from 00 to 30) representing the subset $\mathcal{R}$ of symbols (bytes) employed to represent the compressed or cyphered data.

III. Method Description

Our approach is focused on making entropy analysis resilient to the attacks described in Section II. To this end, we propose a new method for the calculation of binary randomness based on entropy profiles and fine-grained analysis. In addition, we combine the information provided by byte histograms to ignore byte insertion attacks.

A. Entropy profiles and entropy surface

Coarse-grained entropy analysis (file or section entropy) can provide a general overview of the entropy of a file, but as we have seen, in some cases it can bring to wrong conclusions about the real contents of the file. To face this limitation, we propose a method for calculating entropy profiles and for measuring the Entropy Surface in such a way that the value obtained is more resilient to these attacks.

1) **File division**. The strategy adopted to measure entropy divides the file in overlapping regions. Each region is defined by its size s in bytes and the offset o of its first byte with respect to the first byte of the previous region. By setting an offset $o = s/2$ it is possible to cover the executable with overlapping regions.
2) **Entropy**. Once the executable is divided into regions, we calculate the entropy of each region independently, obtaining an entropy profile that provides us with a visual representation (shown in Figure 2) of the randomness of each part of the executable.
3) **Entropy Surface Over a Threshold (ESOT)**. In order to decide whether the executable is packed or not, we have to compute a value from the obtained profile. This value, compared to a surface threshold (T_s), allows us to classify a file into 2 categories: packed or not-packed. To this end, we calculate the ESOT, defined as $ESOT = \sum_{i=0}^{n} A(R_i)$, where n is the number of regions, R_i is the i^{th} region of the file and

$$A(x) = \begin{cases} 0 & \text{if } H(x) \leq T_e \\ H(x) - T_e & \text{if } H(x) > T_e \end{cases}$$

where T_e is the entropy threshold selected. In this manner, it is necessary to stablish 2 thresholds: T_e defines a line which divides the profile into 2 regions, and T_s is compared against the $ESOT$ value calculated, which is equivalent to the area of the upper region. If ESOT surpasses T_s the executable is considered as packed. This value, on the opposite to the value obtained by a coarse-grained analysis (entropy of the whole file), is affected by highly entropic areas in the executable that might not influence the final result when simpler techniques are applied.

B. Byte histogram combination

The histogram of the executable provides the malware analyst with some hints about the byte distribution of the executable. Our approach tries to combine this valuable information to generate a more representative entropy profile, solving the first attack described in Section II.

To this end, we apply the k-medoids clustering algorithm to generate 2 clusters of byte frequencies. This algorithm is a variation of k-means and divides a set of points into k clusters that, on the contrary to k-means, are centred around points in the set. The result of this process is shown in the histogram represented in Figure 1. The light bars represent the bytes whose frequency is higher, while the dark bars represent the bytes whose frequency is lower. This technique allows us to identify the bytes which may be inserted with the aim to reduce entropy of the executable (attack 1), and discard them when measuring entropy. In this case, entropy is calculated for a source alphabet with a lower number of symbols. Nevertheless, our approach could discard bytes that, for any reason, are more frequent either in packed or not-packed software not subject to any entropy attack. To prevent this, we established a size threshold for the cluster composed of highly frequent bytes. In this way, if the number of points in the cluster surpasses the threshold, the bytes belonging to it are not discarded from the entropy calculation. In Section IV we evaluate the different thresholds that can be applied.

Anyhow, the maximum value for the entropy of a region with a reduced source alphabet will be lower than 8 (value for 256 different symbols). In order to balance the entropy profiles calculated for binaries with a different number of bytes discarded, it is necessary to weight the entropy by a value inversely proportional to the maximum possible entropy for the reduced alphabet, obtaining values ranging from 0 to 8: $H'(x) = H(x) * \log_b(N)/\log_b(N - |\mathcal{B}|)$, where $H(x)$ is the calculated entropy, N is the number of symbols in the source alphabet, and $|\mathcal{B}|$ is the number of bytes ignored in the calculation of $H(x)$. This normalization is necessary to obtain balanced values for T_e and T_s for the different files, independently of the number of bytes ignored in the entropy profile calculation process.

IV. Empirical Validation

To validate our approach, we performed an experiment with a dataset composed of 1,000 not-packed executables, 1,000 executable files protected with known packers, and 1,000 samples protected with custom packers. Initially, we gathered 1,000 malware executables from VxHeavens and 1,000 goodware executables from a clean installation of Microsoft Windows XP, and checked the samples with PEiD to assure that they were not-packed. Afterwards, we selected 1,000 (500 goodware and 500 malware) as not-packed executables, and packed the other 1,000 with 10 different packers: Armadillo, ASProtect, FSG, MEW, PackMan, RLPack, SLV, Telock, Themida and UPX. The other 1,000 executables were Zeus malware samples packed by custom made packers.

The experiment performed consisted of 2 phases:

1) **Byte histogram and entropy profile computation** for each executable sample. Once the byte histogram is extracted, the k-medoid algorithm is applied. If the cluster of highly frequent bytes has a number of elements lower than a certain parameter, the bytes are discarded during the computation of the entropy profile.
2) **Variable parameter computation**. Once the entropy profiles are calculated, 2 parameters (i.e., entropy threshold T_e and surface threshold T_s) must be optimised in order to establish a limit value. When a sample surpasses that threshold, it can be labelled as packed. The optimisation process must adjust the parameters to minimise the samples incorrectly classified.

A. Experiment parameters

The experiment depends on certain parameters that must be optimised in order to maximise the accuracy of the system when classifying between packed and not-packed software. Other parameters must be set to concrete values. Table I shows the different experimental configurations tested. The parameter *Maximum Bytes to Ignore* determines the maximum size allowed for the cluster of bytes to be ignored. A value of 0 means that no byte will be ignored.

TABLE I
Parameters for each experimental configuration.

Parameter	Fixed/Variable	Values
Region size	Fixed	128,256,512
Region offset	Fixed	64,128,256
Max. bytes to ignore	Fixed	0,8,16,24
Entropy threshold T_e	Variable	-
Surface threshold T_s	Variable	-

For the optimisation of the variable parameters in each experimental configuration, we employed the AForge[5] library for the .NET Platform. The genetic algorithm employed had the following configuration: each *chromosome* is composed of 2 decimal values (i.e., entropy threshold T_e and surface threshold T_s). The evaluation function that must be maximised, measures the samples correctly classified (accuracy) for the specified thresholds. The mutation function sets one of the two parameters to a random value. The crossover function swaps the T_e parameter of the two chromosomes. The initial chromosome population is 50, the percentage of new chromosomes completely substituted in each generation is 30%. The mutation rate is 10% and the crossover rate is 75%. Finally, the chromosome selection rule is Roulette Wheel, which randomly selects the remaining chromosomes, weighing up those that return a higher evaluation result (classification accuracy).

B. Results

To compare our method, we measured the file entropy in a coarse-grained style, and established a threshold to classify samples into packed or not-packed software. To this end, we employed the C4.5 algorithm and configured it to generate only two branches to divide the space of executables into two categories. The entropy limit calculated by the algorithm was 6.608 and the accuracy obtained in the classification was 0.861.

The results obtained (shown in Table II) measure the performance of our approach in terms of False Positive Rate (FPR), False Negative Rate (FNR), and Accuracy, defined

[5]Available online: http://www.aforgenet.com/

as $Acc = TP + TN/Total$, where TP is the number of files correctly classified as packed, TN is the number of files correctly classified as not packed, and $Total$ is the total number of files. The best results were obtained for regions of 128 bytes with an offset of 64 bits and a limit of 16 bytes to ignore: an accuracy of 0.952 was achieved. It is noticeable that the experimental rounds which do not apply the technique based on byte ignoring proposed in Section III-B achieved inferior results than the ones that apply it, specially for FPR.

TABLE II
RESULTS FOR THE DIFFERENT EXPERIMENTAL CONFIGURATIONS.

Region size and offset	Max. bytes to ignore	T_e	T_s	Acc.	FPR	FNR
Coarse-grained analysis		-	-	0.861	0.171	0.075
128, 64	0	1.757	3.040	0.935	0.073	0.060
256, 128	0	2.546	2.532	0.926	0.142	0.040
512, 256	0	2.439	2.866	0.919	0.148	0.048
128, 64	8	3.628	1.803	0.949	0.068	0.043
256, 128	8	2.917	2.727	0.942	0.080	0.046
512, 256	8	3.301	1.680	0.952	0.075	0.034
128, 64	16	2.399	2.505	**0.952**	**0.068**	**0.038**
256, 128	16	3.581	1.834	0.946	0.070	0.046
512, 256	16	2.757	2.861	0.944	0.080	0.044
128, 64	32	4.471	0.744	0.949	0.069	0.041
256, 128	32	3.691	1.734	0.945	0.075	0.044
512, 256	32	3.561	2.227	0.941	0.065	0.056

V. Discussion and Conclusions

Malware writers design and implement methods to bypass current filters and anti-virus systems. To achieve this, they modify the values of certain features that have been considered relevant to classify samples for a long time. Entropy is a measure commonly used to determine whether an executable is packed or not. This kind of file classification can be a previous filtering step to a time-consuming dynamic unpacking process.

In this paper we provide a method for measuring executable entropy more resilient to techniques focused on reducing randomness. In addition, we document some of the attacks found on real Zeus family malware samples. We test the method by establishing a threshold for our alternative entropy measure, obtaining better results than classic file entropy measure.

Nevertheless, there are some aspects that should be tackled in future work. First, we describe an attack based on the use of a reduced source alphabet, and propose a method for measuring entropy in such conditions, but we do not propose any method to identify which parts of the executable, if any, implement this kind of attack. The identification of these regions would help malware analysts to measure entropy more reliably and to find the regions that hide the real code.

Secondly, binaries are rarely classified using entropy as a single feature. It would be interesting to combine the representation proposed here with other commonly used features to train machine-learning classifiers.

Finally, the attacks described are only an example of what malware writers can do to bypass current anti-malware solutions. These kind of attacks should be studied in order to enhance existing approaches and to design systems more resilient to future or unknown techniques.

Acknowledgment

We would like to acknowledge S21Sec for the malware samples described in this paper. This research was partially supported by the Basque Government under a pre-doctoral grant given to Xabier Ugarte-Pedrero.

References

[1] Faster Universal Unpacker. [Online]. Available: http://code.google.com/p/fuu/

[2] Wyke J., "What is zeus? technical paper." [Online]. Available: http://www.sophos.com/en-us/why-sophos/our-people/technical-papers/what-is-zeus.aspx

[3] M. Morgenstern and H. Pilz, "Useful and useless statistics about viruses and anti-virus programs," in *Proceedings of the CARO Workshop*, 2010. [Online]. Available: http://www.f-secure.com/weblog/archives/Maik_Morgenstern_Statistics.pdf

[4] L. Böhne, "Pandoras bochs: Automatic unpacking of malware," Ph.D. dissertation, 2008.

[5] K. Babar and F. Khalid, "Generic unpacking techniques," in *Proceedings of the 2^{nd} International Conference on Computer, Control and Communication (IC4)*. IEEE, 2009, pp. 1–6.

[6] Data Rescue, "Universal PE Unpacker plug-in." [Online]. Available: http://www.datarescue.com/idabase/unpack_pe

[7] J. Stewart, "Ollybone: Semi-automatic unpacking on ia-32," in *Proceedings of the 14^{th} DEF CON Hacking Conference*, 2006.

[8] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Proceedings of the 2006 Annual Computer Security Applications Conference (ACSAC)*, 2006, pp. 289–300.

[9] M. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM workshop on Recurring malcode*, 2007, pp. 46–53.

[10] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC)*, 2007, pp. 431–441.

[11] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A Framework for Enabling Static Malware Analysis," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2008, pp. 481–500.

[12] A. Danielescu, "Anti-debugging and anti-emulation techniques," *CodeBreakers Journal*, vol. 5, no. 1, 2008. [Online]. Available: http://www.codebreakers-journal.com/

[13] S. Cesare, "Linux anti-debugging techniques, fooling the debugger." [Online]. Available: http://vx.netlux.org/lib/vsc04.html

[14] L. Julus, "Anti-debugging in WIN32," 1999. [Online]. Available: http://vx.netlux.org/lib/vlj05.html

[15] M. Shafiq, S. Tabish, and M. Farooq, "PE-Probe : Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables," in *Proceedings of the 2009 Virus Bulletin Conference (VB)*, 2009, pp. 1–10.

[16] R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 301–310.

[17] X. Ugarte-Pedrero, I. Santos, and P. G. Bringas, "Structural feature based anomaly detection for packed executable identification," in *Proceedings of the 4^{th} International Conference on Computational Intelligence in Security for Information Systems (CISIS)*, 2011, pp. 231–238.

[18] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P. G. Bringas, "Collective classification for packed executable identification," in *Proceedings of the 8^{th} Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)*, 2011, pp. 23–30.

[19] X. Ugarte-Pedrero, I. Santos, P. Bringas, M. Gastesi, and J. Esparza, "Semi-supervised learning for packed executable detection," in *In Proceedings of the 5th International Conference on Network and System Security (NSS)*, 2011, pp. 342–346.

[20] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 40–45, 2007.

[21] L. Sun, "REFORM: A framework for malware packer analysis using information theory and statistical methods," Ph.D. dissertation, 2010.