UNIVERSITY OF PIRAEUS

Department of Digital Systems

MSC Digital Systems Security

Master Thesis

Use of entropy for malware identification

Ouroumidis Athanasios

Supervisor

Prof. Dadoyan Christoforos

Piraeus, Greece, March 2017

1 Abstract

In today’s world internet has become a necessity not only for businesses but also for a person’s daily life. Communication, trade, information, entertainment and many other functionalities are provided to us by being “online”. However together with the evolution of software to help in our day to day activities, the malicious software came to rise. Starting from being created just for fun and ending up created for financial gain, the IT industry face a huge challenge. Malicious software is being created every day and new variations of the same exploits are found every week. The security industry is on cat and mouse race with malware authors to stay ahead and provide a shield from the dangers on the internet.

However this becomes a bigger challenge as the time goes by because more and more checks need to be done from a security standpoint when checking if a piece of code or executable is safe or not. These checks many times are very clear on the definition and identification of malicious files but many times the checks are vague and do not pose a clear factor on identifying potential threats. One of those checks is the value of entropy on the files that the anti-virus vendors use to identify a possible malware.

On this thesis we will explore the definition of entropy as well a possible process of modifying this value to reach lower levels that could be found on a legitimate software.

2 Acknowledgements

I would like before the beginning of this thesis to take this chance to thank my professor, who contributed to a valuable learning experience during the MSC studies. Taught me new and exciting things about the security industry that helped me in my professional life and will help me further in my career.

I would also like to help my family who was there for me whenever I needed them, supported me with any means they had available during my difficult times and always believed in me. Without them I would not be here.

3 Table of contents

4 Information Theory - Entropy

Information can be thought of as being stored in or transmitted as variables that can take on different values. A variable can be thought of as a unit of storage that can take on, at different times, one of several different specified values, following some process for taking on those values. Informally, we get information from a variable by looking at its value, just as we get information from an email by reading its contents. In the case of the variable, the information is about the process behind the variable. The entropy of a variable is the "amount of information" contained in the variable. This amount is determined not just by the number of different values the variable can take on, just as the information in an email is quantified not just by the number of words in the email or the different possible words in the language of the email. Informally, the amount of information in an email is proportional to the amount of “surprise” its reading causes. For example, if an email is simply a repeat of an earlier email, then it is not informative at all. On the other hand, If say the email reveals the outcome of a cliffhanger election, then it is highly informative. Similarly, the information in a variable is tied to the amount of surprise that value of the variable causes when revealed.

Shannon’s entropy quantifies the amount of information in a variable, thus providing the foundation for a theory around the notion of information.

In the IT world the Values of entropy range from 1 to 7 are used to represent the predictability of the next character or byte in a sequence of characters or bytes.

Values closer to 1 means that the entropy is lower therefore the information we can get is lower. Meaning that we can “guess” with a higher probability the next character or byte in line.

Values closer to 7 means that the entropy is higher therefore the information we can get is higher. Meaning that we can “guess” with a lower probability the next character or byte in line.

5 Portable Executable

The Portable Executable (PE) format is a file format for executables, object code, DLLs, FON Font files, and others used in 32-bit and 64-bit versions of Windows operating systems. The PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data. On NT operating systems, the PE format is used for EXE, DLL, SYS (device driver), and other file types. The Extensible Firmware Interface (EFI) specification states that PE is the standard executable format in EFI environments.[24]

5.1 Basic Structure

A Portable Executable (PE) basically contains two sections, which can be subdivided into several sections. One is Header and the other is Section. The diagram below shows a visualized version of the PE.

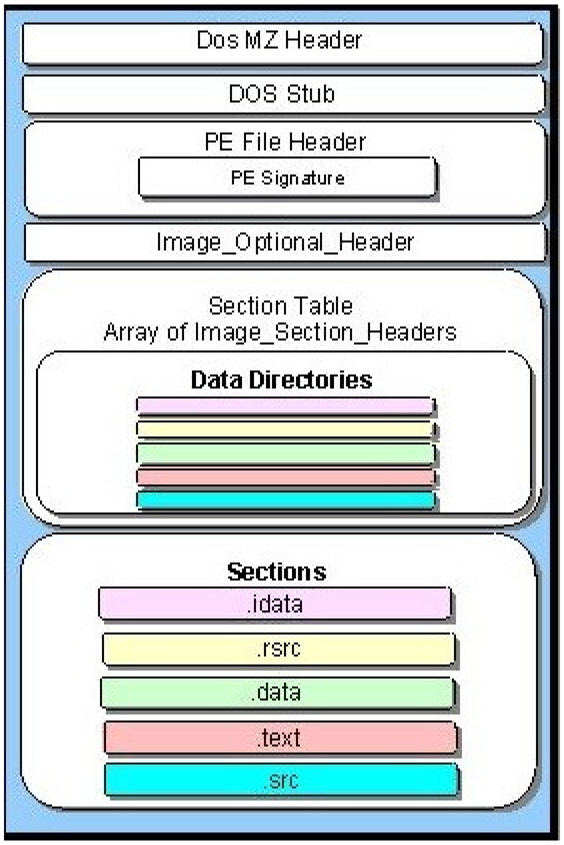


Image 1: PE file format

1. DOS Header

DOS header starts with the first 64 bytes of every PE file. It’s there because DOS can recognize it as a valid executable and can run it in the DOS stub mode.

2. DOS Stub

The DOS stub usually just prints a string, something like the message, “This program cannot be run in DOS mode.”, which is shown when the PE tries to run in DOS mode. It can be a full-blown DOS program. When building applications on Windows, the linker sends instruction to a binary called winstub.exe to the executable file. This file is kept in the address 0x3c, which is offset to the next PE header section.

3. PE File Header

Like other executable files, a PE file has a collection of fields that defines what the rest of file looks like. The header contains info such as the location and size of code,

4. Characteristics

- **Signature:** It only contains the signature so that it can be easily understandable by windows loader. The letters P.E. followed by two 0’s tells everything.
- **NumberOfSections:** This defines the size of the section table, which immediately follow the header.
- **SizeOfOptionalHeader:** This lies between top of the optional header and the start of the section table. This is the size of the optional header that is required for an executable file. This value should be zero for an object file.
- **Characteristics:** This is the characteristic flags that indicate an attribute of the object or image file. It has a flag called Image_File_dll, which has the value 0x2000, indicating that the image is a DLL. It has also different flags that are not required for us at this time
- **Image_Optional_Header:** This optional header contains most of the meaningful information about the image, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information, and so forth. We can see the information in the snapshot below.

5. The Section Table

This table immediately follows the optional header. The location of this section of the section table is determined by calculating the location of the first bytes after header. For that, we have to use the size of the optional header. The number of the array members is determined by NumberOfSections field in the file header (IMAGE_FILE_HEADER) structure. The structure is called IMAGE_SECTION_HEADER.

The number of entries in the section table is given by noofsectionfield in the file header. Each section header has at least 40 bytes of entry. We will discuss some of the important entries below.

- Name: An 8-byte null-padded UTF8 encoding string. This is can be null.
- VirtualSize: The actual size of the section's data in bytes. This may be less than the size of the section on disk.
- SizeOfRawData: The size of section's data in the file on the disk.
- PointerToRawData: This is so useful because it is the offset from the file's beginning to the section's data.
- Characteristics: This flag describes the characteristics of the section.

6. The PE File Section

This section contains the main content of the file, including code, data, resources and other executable files. Each section has a header and body.

An application in Windows NT typically has nine different predefined sections, such as .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug. Depending on the application, some of these sections are used, but not all are used.

The Executable Code: In Windows, all code segments reside in a section called .text section or CODE. We know that windows uses a page-based virtual system, which means having one large code section that is easier to manage for both the OS and application developer. This also called as entry point and thunk table, which points to IAT. We will discuss the thunk table in IAT.

- The .bss represents the uninitialized data for the application.
- The .rdata represents the read-only data on the file system, such as strings and constants.
- The .rsrc is a resource section, which contains resource information of a module. In many cases it shows icons and images that are part of the file's resources.

- The .edata section contains the export directory for an application or DLL. When present, this section contains information about the names and addresses of exported functions. We will discuss these in greater depth later.
- The .idata section contains various information about imported functions, including the import directory and import address table. We will discuss these in greater depth later.

6 Malware

Is a software program developed to perform malicious activities on a computer and refers to a variety of different forms of hostile or intrusive software. There can be any reasons for writing malware varying from simple pranks to organized internet crimes. The early infectious programs were written as pranks. These days, malware is widely used to steal personal, financial, business information. Malware includes all families of viruses, computer worms, Trojans, backdoor, spyware, adware, scareware, ransomware. A brief overview of different types of malware is given below.[25]

- **Virus:** A virus is a type of malware that replicates itself by inserting copies or modified copies of itself into other programs. It is designed to change the way the computer operates. They can live anywhere. It can live on the boot sector. If it lives on the boot sector, it can take control before anything else. It establishes itself before any antivirus software starts or operating system security is enabled. It can also live in the memory. It can enter the computer by any way the user interacts with the computer (i.e. open an email, plug in an external storage device, open a website that is infected, open malicious files etc). They have the ability to reproduce themselves by infecting other files and programs with malicious code. When they are run, they are able to carry out a range of usually malicious actions in the computer or simply annoying. Virus writers constantly modify their software to evade the detection techniques. The prominent methods to evade the detection techniques are encryption, polymorphism and metamorphism. [26][22][15]

- **Polymorphic Virus:** A polymorphic virus has, for all practical purposes, an infinite number of decryptor loop variations that's morphed with each generation. Tremor, for example, has almost six billion possible decryptor loops! Polymorphic viruses clearly can't be detected by listing all the possible combinations. The techniques such as emulation can be used for polymorphic virus detection.[15]

- **Encrypted Virus:** Antivirus software searches for a signature (a specific bit string) for virus detection. The simplest method to hide the virus body is to encrypt it with different encryption keys. As a result of this, the detection of a virus becomes a difficult task. The idea of an encrypted virus is to encrypt the signature in order to evade signature detection. However, it is still possible to search for an encrypted signature too. Thus, the encrypted virus is not a reliable way of evading signature detection. The only part that is constant in the encrypted virus is the decryptor loop. Antivirus software will exploit this fact for detection, so the next logical development is to change the decryptor loop's code with each infection. [15]

- **Metamorphic Virus:** Virus writers modified the malware furthermore to avoid emulation detection. The metamorphic virus is also called as body polymorphic virus. The appearance of the virus changes before infecting any system. The detection of a metamorphic virus is very challenging. The morphed virus has the same functionality but a different structure. Hence the detection of metamorphic virus is difficult.

- **Worm:** A computer worm is a standalone malware computer program that replicates itself in order to spread to other computers. Worms are programs that replicate themselves from system to system without the use of a host file. Unlike viruses, which requires the spreading of an infected host file. Worms replicate themselves damaging files, but can reproduce rapidly, saturating a network and causing it to collapse. [16][27][22]

- **Trojan:** A Trojan is a type of malware which appears to perform a desirable function but instead inserts a malicious payload to the target. An crucial difference between Trojan and a virus is that the Trojan does not replicate itself. They pose as legitimate programs that users know and they intend to use but when these are executed, they install a malicious payload into the target host for various purposes that the trojan author created them. The Trojans have the capacity of deleting files, destroying information on the hard drive, or open a backdoor to the security systems. [27][28][22]

- **Trapdoor/Backdoor:** A trapdoor/backdoor is a program which bypasses the security check in place of personal network or corporate. This allows a malicious user to carry out various actions on the infected computer that can compromise user confidentiality or disrupt the actions that are being carried out. The actions that a backdoor allow malicious users to carry out can be extremely damaging for for the user but also for others in the network. They could allow them to delete files, destroy all the information on the hard disk, capture confidential data and send it out to an external address or open communications ports, allowing remote control of the computer. [15][22]

- **Ransomware:** A type of malicious software that threatens the victim with publishing its private confidential information or blocking the access to its data by encrypting them. To gain access to its data or not publishing the victim's private information to the internet, ransomware authors demand a “ransom”, usually in some kind of cryptocurrency before they provide the victim with the unlocking code for their files. These types of attack come along with a time limit that the victim has to sent the ransom before the data stays forever encrypted or the data are published online.

6.1 Malware Detection Techniques

As malware writers fine-tune their software by making it better to evade signature detection, the anitivirus companies are improving their detection techniques as well.

6.1.1 Signature Based Detection

Signature based detection is a simple and most commonly used technique in antivirus software. They are popular because of accurate detection, simplicity and speed. In signature based detection, the scanner scans each executable and looks for specific string or pattern of bits (signatures). Antivirus software has a database of signatures for different viruses. By comparing the signature, it detects the virus. The disadvantage is that only the known malware can be detected. If the signature is not known, malware cannot be detected. The signature file must be kept up to date. By using simple code obfuscation techniques, malware can easily evade the signature based detection. [29][30]

6.1.2 Anomaly Based Detection

The problem of detecting new malwares in signature based detection can be overcome using anomaly based detection. Heuristic methods are implemented to detect anomalous behavior. This technique comprises of two phases - the training phase and the detection phase. In the training phase, the model is trained with the normal behavior. Anything other than the normal behavior is considered as malicious behavior. However, there can be more false positives in this technique.[18][29]

6.1.3 Heuristics Based Detection

Unlike signature-based detection, which looks to match signatures of files against a database of known malware, heuristic scanning uses rules and/or algorithms to look for commands which may indicate malicious intent. When using this method, some heuristic scanning methods are able to detect malware without needing a signature comparison. This is why most antivirus programs use both signature and heuristic-based methods in combination, in order to catch any malware that may try to evade detection by using obfuscation techniques or any new malware that has not been discovered yet.

7 Malware Classification Methods

7.1 Classification of Malware using Structured Control Flow

Control flow represents the execution path that a program can take. In related research it has been shown that malware can be effectively be characterized by its control flow. The authors have proposed a malware classification system using approximate matching of control flow patterns. The result of distances can be calculated between the control flow signatures and the structured graphs of the malware in the database. The threshold is decided. If the edit distance exceeds a particular threshold, then the binary can be classified as a malicious binary, else it is a benign binary. Control flow is more invariant among polymorphic and metamorphic malware. The research shows that the proposed method could successfully identify variants of malware.[17]

7.2 Behavioral Malware Classification

Classification systems generally fall into one of two categories: Those that rely on features extracted from static files, or those that execute malware and use behavioral features to classify malware. Static approaches sometimes use low-level features such as calls to external libraries, strings, and byte sequences for classification. Other static approaches extract more detailed information from binaries, including sequences of API calls, the graphical representations of control flow. Although the variants in a malware family have different static signatures, they share characteristic behavioral patterns resulting from their common generation machine. It has been already described an automatic classification system that can be trained to accurately identify new variants within known malware families, using observed
similarities in behavioral extracted monitoring live computer hosts. In the feature selection used in, the authors have selected a set of observable features that are easily extracted from live computer hosts, and whose values can be used to infer whether a detected malware sample belongs to particular category or family. [20][17][32][31]

7.3 Instance-based Learner

One of the simplest learning methods is the instance-based (IB) learner. Its concept description is a collection of training examples or instances. Learning, therefore, is the addition of new examples to the collection. An example is found in the collection that is most similar to the unknown and the examples class label is returned as its prediction for the unknown. The authors have used the number of values the two instances have in common as the measure of similarity. In the variation of this method, such as IBk, the k most similar instances are found and the majority vote of their class labels is returned as the prediction. Values for k are typically odd to prevent ties. These are also called as nearest neighbor and k-nearest neighbors.[20]

7.4 Support Vector Machines (SVM)

Support Vector Machines are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification.

7.5 Naïve Bayes

Naïve Bayes is a probabilistic method that has a long history in information retrieval and text classification. It stores as its concept description the prior probability of each class, and the conditional probability of each attribute value given the class. These quantities are estimated by counting in training data the frequency of occurrence of the classes and the attribute values for each class. The Bayes rule is used to compute the posterior probability of each class given an
unknown instance, returning as its prediction the class with highest such value.[20]

7.6 Data mining methods

In related researches the authors have extracted the byte sequences from the executables,
converting these into n -grams, and constructed several classifiers: instance-based learner, Naïve Bayes, decision trees, support vector machines and boosting. They viewed each n-grams as a Boolean attribute that is either present in or absent from the executable. They have shown that the boosted decision trees outperformed the other methods. The following section shows the methods used in their research.[20]

7.7 Decision Trees

The decision trees are built based on the training data. The internal nodes of a decision tree correspond to attributes and leaf nodes correspond to class labels. The performance element uses the attributes and their values of an instance to traverse the tree from the root to a leaf. It predicts the class label of the leaf node. It creates a node, branches, and children for the attribute and its values, removes the attribute from further consideration, and distributes the examples to the appropriate child node. This process repeats recursively until a node contains examples of the same class, at which point, it stores the class label. In an effort to reduce over training, most implementations also prune induced decision trees by removing subtrees that are likely to perform poorly on test data. The malware classification based on the decision trees is very fast and also accurate. The disadvantage of the decision trees is that an error in higher level of the tree may cause an error in the lower part of the tree.[20]

7.8 Boosted Classifiers

Boosting is a method for combining multiple classifiers. A set of weighted models are produced by iteratively learning a model from a weighted dataset. The generated model is then evaluated. The dataset is re-weighted based upon the model's performance. The authors have provided a method of detecting unknown malicious code in executables using machine learning. They have extracted byte sequences from the executables, converted these into n-grams, and constructed several classifiers: naïve Bayes, boosted SVMs and boosted decision trees. The results of their experiments have shown that the boosted decision trees outperformed other methods and achieved a true-positive rate of 0.98 and a false-positive rate of 0.0. [19][20]

7.9 Structural Entropy

The method of structural entropy lies in the static analysis of files and produces a similarity measure, i.e. evaluates to which extent the two files can be considered similar. The only thing of importance is file structure, that is, the order of its distinctive code and data areas. The entropy measure provides a sort of signature of a file, by computing the distribution of bytes within the file. The assumption is that different malware samples of the same family have a similar order of code and data areas; as a matter of fact each area may be characterized not only by its length, but also by its homogeneity. Authors in related research identify as structural entropy this characteristic of an application. The approach consists of using discrete wavelet transform (DWT) for the segmentation of files into segments of different entropy levels and using edit distance between sequence segments to determine the similarity of the files. The method comprises two steps: file segmentation and sequence comparison. The first step splits each file into segments of varying entropy levels using wavelet analysis applied to raw entropy measurements.[2][3]

7.10 Hidden Markov Model Based Detection

Hidden Markov models (HMMs) are generally used for statistical pattern analysis. They can be used in speech recognition, malware detection and biological sequence analysis. The following sections give an overview of the introduction to HMM and its usage in detection of malware.

A statistical model that has states and known probabilities of the state transitions is called a Markov model. In such a Markov model, the states are visible to the observer. In contrast, a hidden Markov model (HMM) has states that are not directly observable. HMM is a machine learning technique. HMM acts as a state machine. Every state is associated with a probability distribution for observing a set of observation symbols. The transition between the states have fixed probabilities. We can train an HMM using the observation sequences to represent a set of data. We can match an observation sequence against a trained HMM to determine the probability of seeing such a sequence. If the probability is high, the observation sequence is similar to the training sequence.

When an HMM is trained, it can be used to distinguish between a malware and a benign file. There is a lot of previous work done on the use of HMM for malware detection. The dataset is tested against the trained models. There is a range of values of scores for which the scores of the malware and the benign files do not overlap. This is known as threshold. Using this threshold, the malware can be distinguished from the benign files.[1][21]

8 Malware Analysis Techniques

8.1 Malware Static Analysis

Basic static analysis consists of examining the executable file without viewing the actual instructions. Basic static analysis can confirm whether a file is malicious, provide information about its functionality, and sometimes provide information that will allow you to produce simple network signatures. Basic static analysis is straightforward and can be quick, but it's largely ineffective against sophisticated malware, and it can miss important behaviors.[7]

8.1.1 Static Analysis Techniques

- File Fingerprinting: During the File Fingerprinting for every file under investigation a cryptographic hash value will be computed.[7]
- Virus Scanning: The files under investigation will be scanned with different antivirus vendors to identify any possible warnings.[7]
- Analyzing memory artifacts: During this process memory artifacts will be analyzed such as Ram dump, page file.sys hiberfile.sys, so that they can be identified any possible rogue processes.[7]
- Packer Detection: Almost always the malware will be packed with some kind of packer. The files will need to be analyzed and their packer will need to be identified.[7]
- Disassembly: Many times malwares use dynamic linking in their code. The Dependencies can be analyzed using various tools while it can also be done by disassembling the executable.[7]

8.2 Malware Dynamic Analysis

8.2.1 Function Call Monitoring

Typically, a function consists of code that performs a specific task, such as calculating the factorial value of a number or creating a file. While the use of functions can result in easy code re-usability, and easier maintenance, the property that makes functions interesting for program analysis is that they are commonly used to abstract from implementation details to a semantically richer representation. For example, the particular algorithm which a sort function implements might not be important as long as the result corresponds to the sorted input. When it comes to analyzing code, such abstractions help gain an overview of the behavior of the program. One possibile way to monitor what functions are called by a program is to intercept these calls. The process of intercepting function calls is called hooking. The analyzed program is manipulated in a way so that, in addition to the intended function, a hook function is invoked. This hook function is responsible for implementing the required analysis functionality, such as recording its invocation to a log file, or analyze input parameters.[7]

8.2.2 Function Parameter Analysis

While function parameter analysis in static analysis tries to infer the set of possible parameter values or their types in a static manner, dynamic function parameter analysis focuses on the actual values that are passed when a function is invoked. The tracking of parameters and function return values enables the correlation of individual function calls that operate on the same object. For example, if the return value (a file-handle) of a CreateFile system call is used in a subsequent WriteFile call, such a correlation is obviously given. Grouping function calls into logically coherent sets provides detailed insight into the program’s behavior from a different, object-centric, point-of-view.[7]

8.2.3 Information Flow Tracking

An orthogonal approach to the monitoring of function calls during the execution of a program, is the analysis on how the program processes data. The goal of information flow tracking is to shed light on the propagation of “interesting” data throughout the system while a program manipulating this data is executed. In general, the data that should be monitored is specifically marked (tainted) with a corresponding label. Whenever the data is processed by the application, its taint-label is propagated. Assignment statements, for example, usually propagate the taint-label of the source operand to the target. Besides the obvious cases, policies have to be implemented that describe how taint-labels are propagated in more difficult scenarios. Such scenarios include the usage of a tainted pointer as the base address when indexing to an array or conditional expressions that are evaluated on tainted values.[7]

8.2.4 Instruction Trace

A valuable source of information for an analyst to understand the behavior of an analyzed sample is the instruction trace. That is, the sequence of machine instructions that the sample executed while it was analyzed. While commonly cumbersome to read and interpret, this trace may contain important information not represented in a higher level abstraction (e.g., analysis report of system and function calls).[7]

8.2.5 Autostart Extensibility Points

Autostart extensibility points (ASEPs) define mechanisms in the system that allow programs to be automatically invoked upon the operating system boot process or when an application is launched. Most malware components try to persist during reboots of an infected host by adding themselves to one of the available ASEPs. It is, therefore, of interest to an analyst to monitor such ASEPs when an unknown sample is analyzed.[7]

9 Testing Methodology

9.1 File Entropy

The entropy of a file is also used from antivirus vendors to identify potential malwares. When files are packet or encrypted their entropy is increased because of the compression algorithm or the packer.

Malware authors that wants to avoid being detected will encrypt their malwares therefore increasing their entropy.

On the following sections of the paper we will see how it's possible to alter the overall entropy of a file but also alter the overall entropy of a section of a file that contains malware code.

9.2 File entropy Manipulation

For our experiment we will need to calculate the entropy of a file in 2 different ways:

The overall entropy of a file. We will calculate the file entropy by treating the file like a long sequence of bytes ignoring its structure. This test will be done by using an open source tool name "ent" (available in github), which the only thing that it does is measuring the entropy of a file regardless of the type we will also use PEid which is a tool that is used to identify if a file is packed, the type of packer and the entropy of a file.

The Section entropy of a file: We will calculate the section entropy, by calculating the entropy of the different sections in the PE (.text, .data, .bss etc) . For this test we will use python code and the "pe" library for python that is already available in github as well as a software called DIE (Detect It Easy) which among other things it has the functionality of calculating the section entropy of a file, which will be used to verify our results.

For both test cases we will use python code to add a section to a PE file. The new section will contain either a long sequence of "NOP" instructions, a shellcode or a shellcode with NOP instructions before and after the shellcode. This will help us test how it will affect the overall entropy and the section entropy of the file.

9.2.1 Methodology Steps

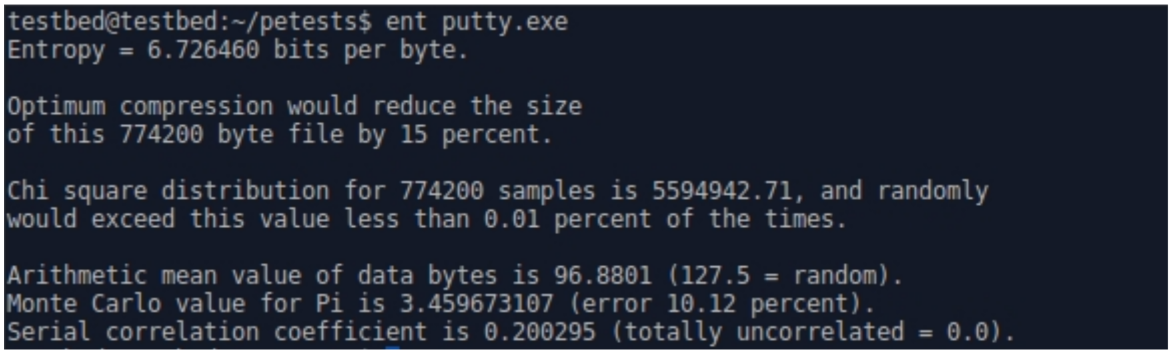
The test steps will be as follows:

1. Creating a new section with NOPs on a legitimate PE.
2. Creating a new section with a shellcode on a legitimate PE
3. Creating a new section with a shellcode that will contain NOPs before and after the shellcode

9.2.2 Entropy of test PE

For testing purposes we will use the putty.exe PE that is free to download and is used very often for ssh connections from windows machines.

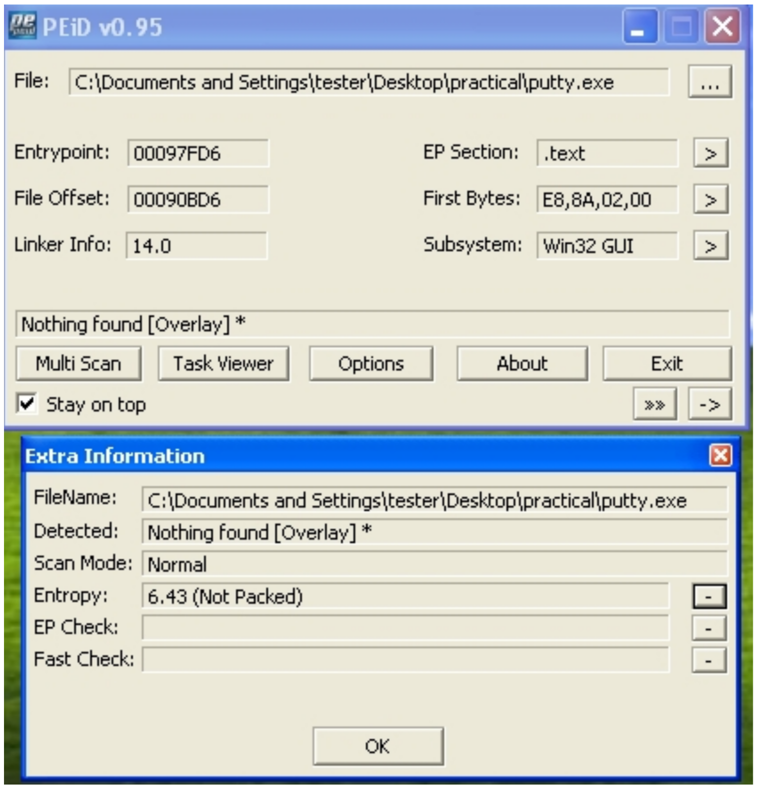
After Downloading putty from its website (https://putty.org) we first test its entropy.



```
testbed@testbed:~/petests$ ent putty.exe
Entropy = 6.726460 bits per byte.

Optimum compression would reduce the size
of this 774200 byte file by 15 percent.

Chi square distribution for 774200 samples is 5594942.71, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 96.8801 (127.5 = random).
Monte Carlo value for Pi is 3.459673107 (error 10.12 percent).
Serial correlation coefficient is 0.200295 (totally uncorrelated = 0.0).
```

Image 2: putty.exe (ent entropy)

As it can be seen from the image the entropy of putty.exe is 6.726460.
If we recall the information theory the entropy values range from 1 till 7, with 7 representing high randomness or very low possibility of guessing the next character in line.
Such high values can be found also in packed executables or compressed.
We test the entropy values both with PEid and DIE tools to verify the result.



File: C:\Documents and Settings\tester\Desktop\practical\putty.exe

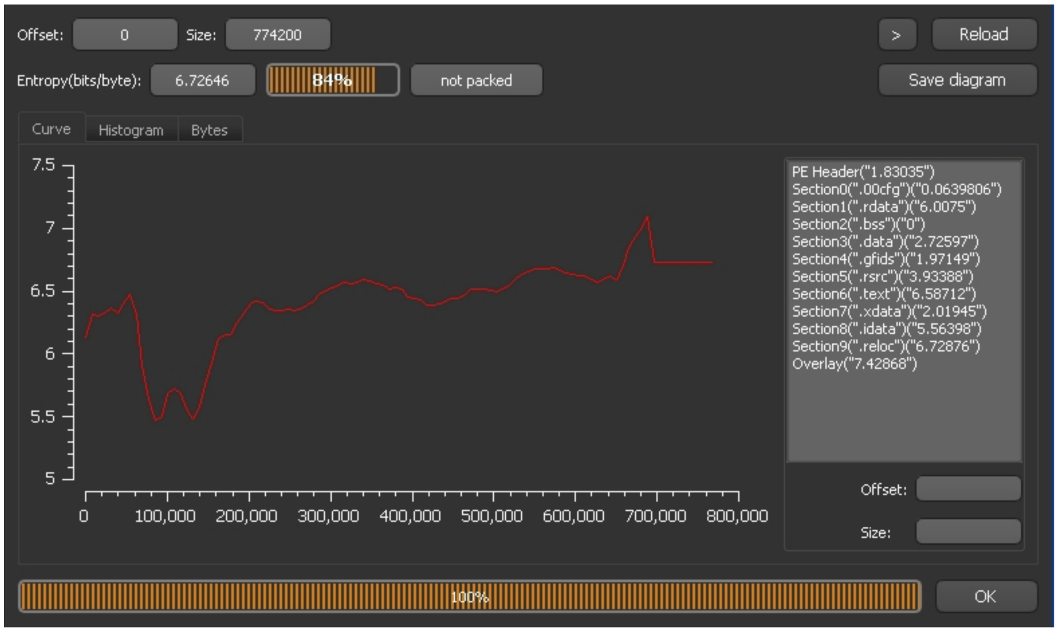
Entrypoint: 00097FD6
EP Section: .text
File Offset: 00090BD6
First Bytes: E8,8A,02,00
Linker Info: 14.0
Subsystem: Win32 GUI

Nothing found [Overlay] *

Extra Information

FileName: C:\Documents and Settings\tester\Desktop\practical\putty.exe
Detected: Nothing found [Overlay] *
Scan Mode: Normal
Entropy: 6.43 (Not Packed)
EP Check:
Fast Check:

Image 3: putty.exe (PEiD Entropy)



Offset: 0 Size: 774200
Entropy(bits/byte): 6.72646 84% not packed

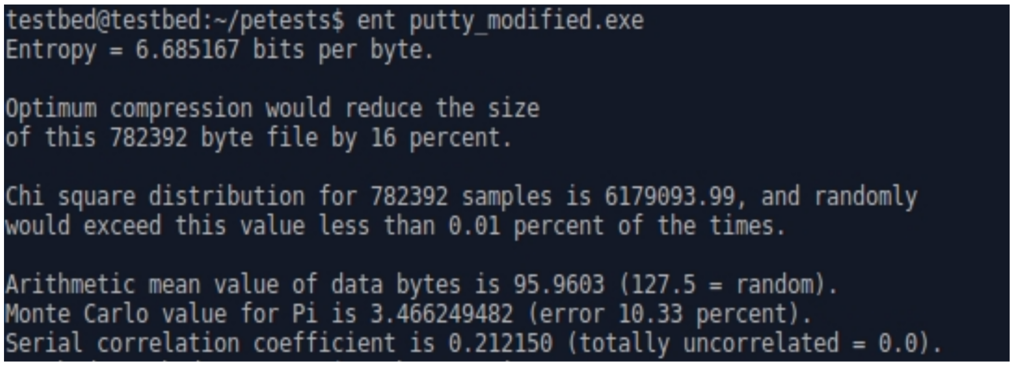
```
PE Header("1.83035")
Section0(".00cfg")("0.0639806")
Section1(".rdata")("6.0075")
Section2(".bss")("0")
Section3(".data")("2.72597")
Section4(".gfids")("1.97149")
Section5(".rsrc")("3.93388")
Section6(".text")("6.58712")
Section7(".xdata")("2.01945")
Section8(".idata")("5.56398")
Section9(".reloc")("6.72876")
Overlay("7.42868")
```

Image 4: putty.exe (DIE entropy)

As we can see “ent” and “DIE” tools report the same entropy values, unlike PEid which has a slightly lower value.
This can be attributed to the possible different algorithms between those tools.

9.2.3 Adding new section to test PE

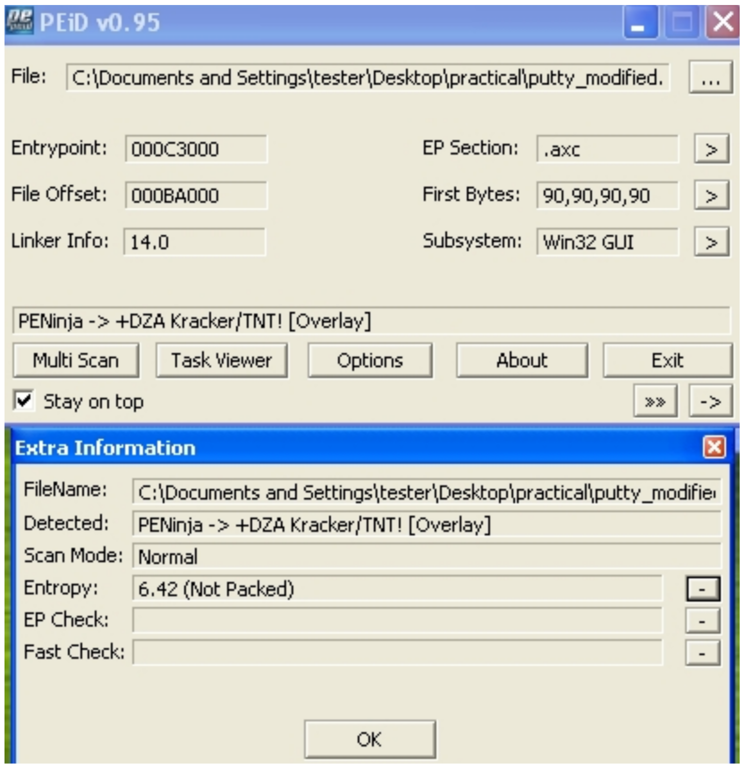
Next step will be to create a new section in putty.exe with the name .axc that will contain a long sequence of NOP instructions (1274, the value was selected to be big enough so that it can fill up the section as much as possible) and we will again compare it with the same tools but also we will use a tool called PE view to take a look inside the code of the new section that we will create.



```
testbed@testbed:~/petests$ ent putty_modified.exe
Entropy = 6.685167 bits per byte.

Optimum compression would reduce the size
of this 782392 byte file by 16 percent.

Chi square distribution for 782392 samples is 6179093.99, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 95.9603 (127.5 = random).
Monte Carlo value for Pi is 3.466249482 (error 10.33 percent).
Serial correlation coefficient is 0.212150 (totally uncorrelated = 0.0).
```

Image 5: Modified putty.exe (ent entropy)

We can see here that the new section resulted in a slight drop of entropy of about 0.04. Such small result is expected as the file is very big (more than 700KB) compared to our section (~1KB), which is not enough to create a big difference.



File: C:\Documents and Settings\tester\Desktop\practical\putty_modified.

Entrypoint: 000C3000
EP Section: .axc
File Offset: 000BA000
First Bytes: 90,90,90,90
Linker Info: 14.0
Subsystem: Win32 GUI

PENinja -> +DZA Kracker/TNT! [Overlay]

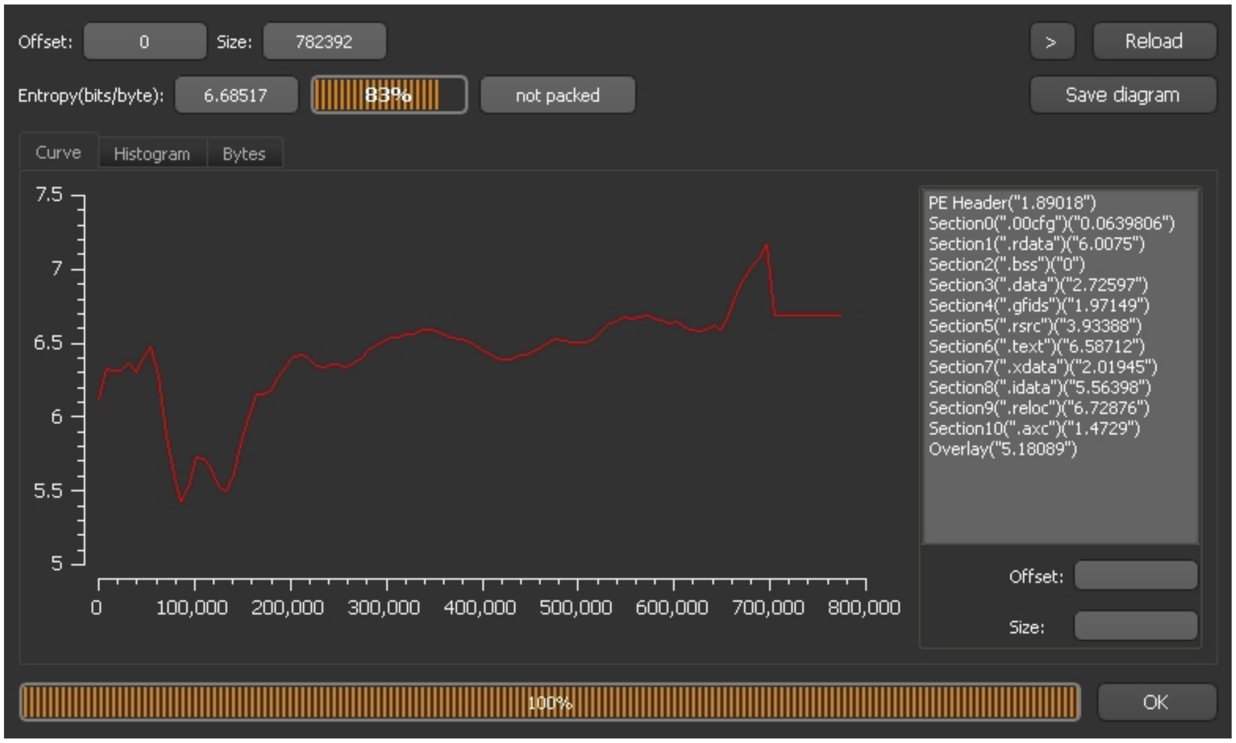
[✓] Stay on top

Extra Information

FileName: C:\Documents and Settings\tester\Desktop\practical\putty_modifie
Detected: PENinja -> +DZA Kracker/TNT! [Overlay]
Scan Mode: Normal
Entropy: 6.42 (Not Packed)
EP Check:
Fast Check:

Image 6: Modified putty.exe (PEiD entropy)

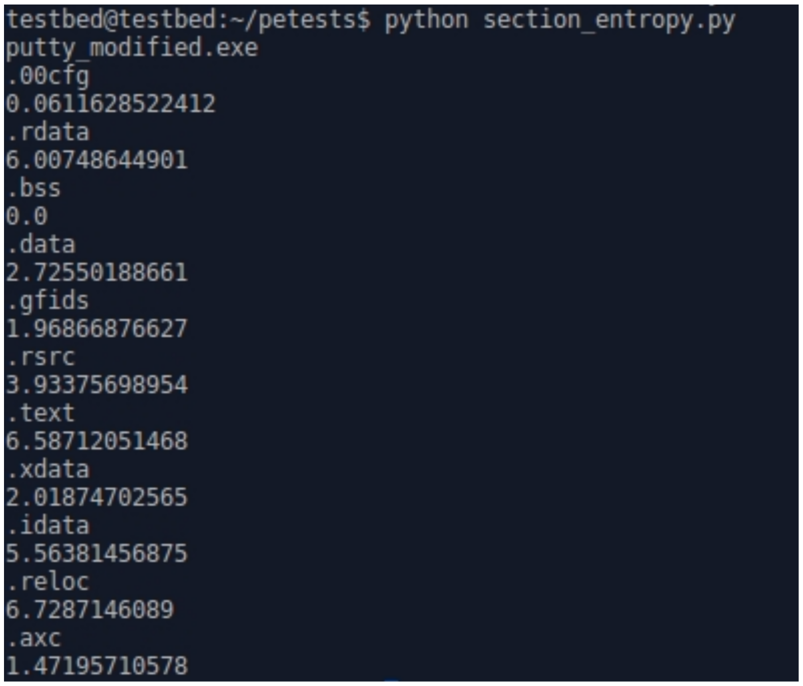
We can confirm as well from PEid that the entropy dropped in this case only a 0.01.



```
PE Header("1.89018")
Section0(".00cfg")("0.0639806")
Section1(".rdata")("6.0075")
Section2(".bss")("0")
Section3(".data")("2.72597")
Section4(".gfids")("1.97149")
Section5(".rsrc")("3.93388")
Section6(".text")("6.58712")
Section7(".xdata")("2.01945")
Section8(".idata")("5.56398")
Section9(".reloc")("6.72876")
Section10(".axc")("1.4729")
Overlay("5.18089")
```

Image 7: Modified putty.exe (DIE entropy)

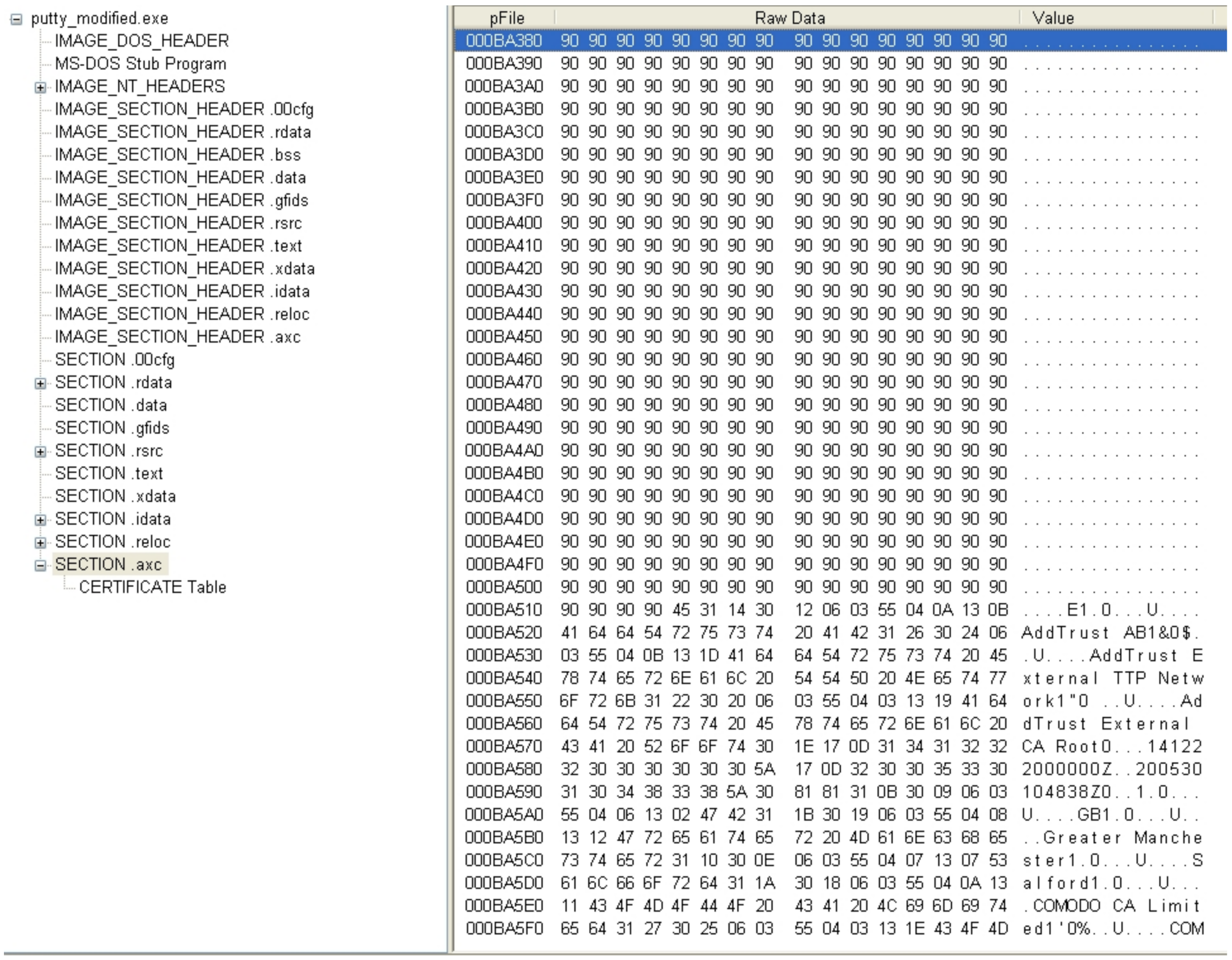
Here we can see not only the entropy that is identical to the value of "ent" but we can also see all the sections along with the new section entropy that we added and its only 1.4729.



```
testbed@testbed:~/petests$ python section_entropy.py
putty_modified.exe
.00cfg
0.0611628522412
.rdata
6.00748644901
.bss
0.0
.data
2.72550188661
.gfids
1.96866876627
.rsrc
3.93375698954
.text
6.58712051468
.xdata
2.01874702565
.idata
5.56381456875
.reloc
6.7287146089
.axc
1.47195710578
```

Image 8: Modified putty.exe (section entropy)

With the python code we can verify our findings about the new section and its entropy (1.4719)



```
putty_modified.exe
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
  IMAGE_SECTION_HEADER .00cfg
  IMAGE_SECTION_HEADER .rdata
  IMAGE_SECTION_HEADER .bss
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .gfids
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .xdata
  IMAGE_SECTION_HEADER .idata
  IMAGE_SECTION_HEADER .reloc
  IMAGE_SECTION_HEADER .axc
  SECTION .00cfg
  SECTION .rdata
  SECTION .data
  SECTION .gfids
  SECTION .rsrc
  SECTION .text
  SECTION .xdata
  SECTION .idata
  SECTION .reloc
  SECTION .axc
    CERTIFICATE Table
```

```
pFile     Raw Data                                          Value
000BA380  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA390  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA3A0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA3B0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA3C0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA3D0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA3E0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA3F0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA400  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA410  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA420  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA430  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA440  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA450  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA460  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA470  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA480  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA490  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA4A0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA4B0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA4C0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA4D0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA4E0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA4F0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA500  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA510  90 90 90 90 45 31 14 30  12 06 03 55 04 0A 13 0B  ....E1.0...U....
000BA520  41 64 64 54 72 75 73 74  20 41 42 31 26 30 24 06  AddTrust AB1&0$.
000BA530  03 55 04 0B 13 1D 41 64  64 54 72 75 73 74 20 45  .U....AddTrust E
000BA540  78 74 65 72 6E 61 6C 20  54 54 50 20 4E 65 74 77  xternal TTP Netw
000BA550  6F 72 6B 31 22 30 20 06  03 55 04 03 13 19 41 64  ork1"0 ..U....Ad
000BA560  64 54 72 75 73 74 20 45  78 74 65 72 6E 61 6C 20  dTrust External 
000BA570  43 41 20 52 6F 6F 74 30  1E 17 0D 31 34 31 32 32  CA Root0...14122
000BA580  32 30 30 30 30 30 30 5A  17 0D 32 30 30 35 33 30  2000000Z..200530
000BA590  31 30 34 38 33 38 5A 30  81 81 31 0B 30 09 06 03  104838Z0..1.0...
000BA5A0  55 04 06 13 02 47 42 31  1B 30 19 06 03 55 04 08  U....GB1.0...U..
000BA5B0  13 12 47 72 65 61 74 65  72 20 4D 61 6E 63 68 65  ..Greater Manche
000BA5C0  73 74 65 72 31 10 30 0E  06 03 55 04 07 13 07 53  ster1.0...U....S
000BA5D0  61 6C 66 6F 72 64 31 1A  30 18 06 03 55 04 0A 13  alford1.0...U...
000BA5E0  11 43 4F 4D 4F 44 4F 20  43 41 20 4C 69 6D 69 74  .COMODO CA Limit
000BA5F0  65 64 31 27 30 25 06 03  55 04 03 13 1E 43 4F 4D  ed1'0%..U....COM
```

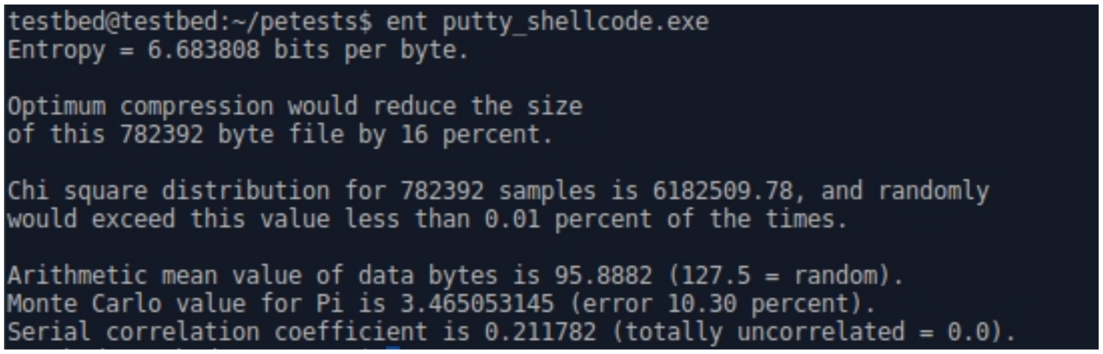
Image 9: Modified putty.exe (PEview sections)

The results from both “DIE” and the written python code resulted in the new section having an entropy of around 1.4 instead of lower that one would expect. This is expected as the screenshot above shows at the end of the section there are garbage values that were added at the moment of the creation, thus resulting in an entropy higher than expected. If the whole section however was filed only with NOPs the entropy value would be lower.

9.2.4 Adding section with shellcode to test PE

Next step will be to add a small shellcode to the test PE and test its overall and section entropy.

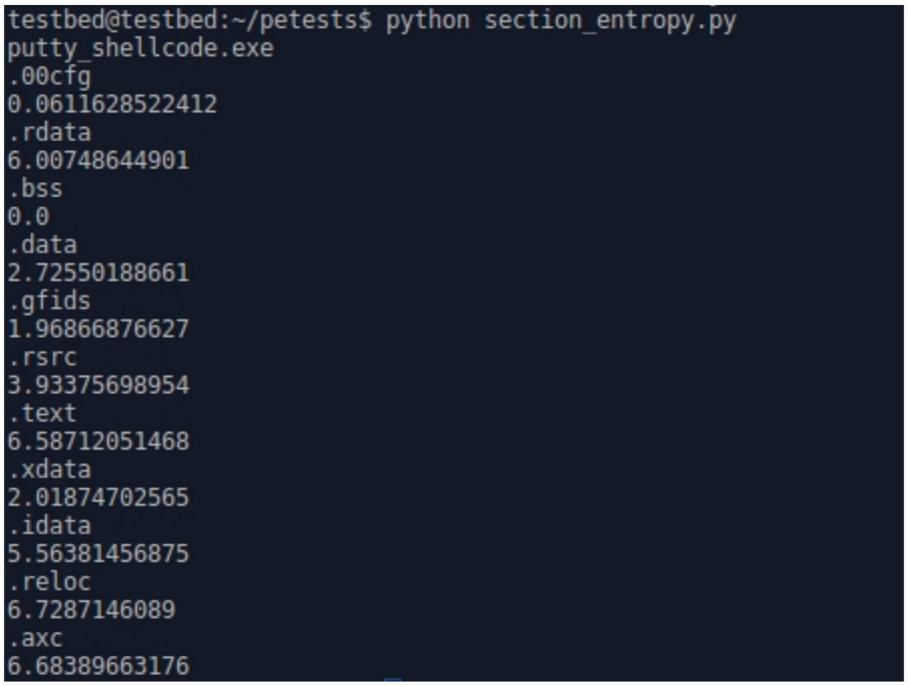
The shellcode that we are going to use has been generated from msfvenom for Microsoft Windows and its functionality is a simple message box popup. For demonstration purposes we don’t need a bigger or more sophisticated shellcode.



```
testbed@testbed:~/petests$ ent putty_shellcode.exe
Entropy = 6.683808 bits per byte.

Optimum compression would reduce the size
of this 782392 byte file by 16 percent.

Chi square distribution for 782392 samples is 6182509.78, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 95.8882 (127.5 = random).
Monte Carlo value for Pi is 3.465053145 (error 10.30 percent).
Serial correlation coefficient is 0.211782 (totally uncorrelated = 0.0).
```

Image 10: putty.exe - shellcode in section (ent entropy)

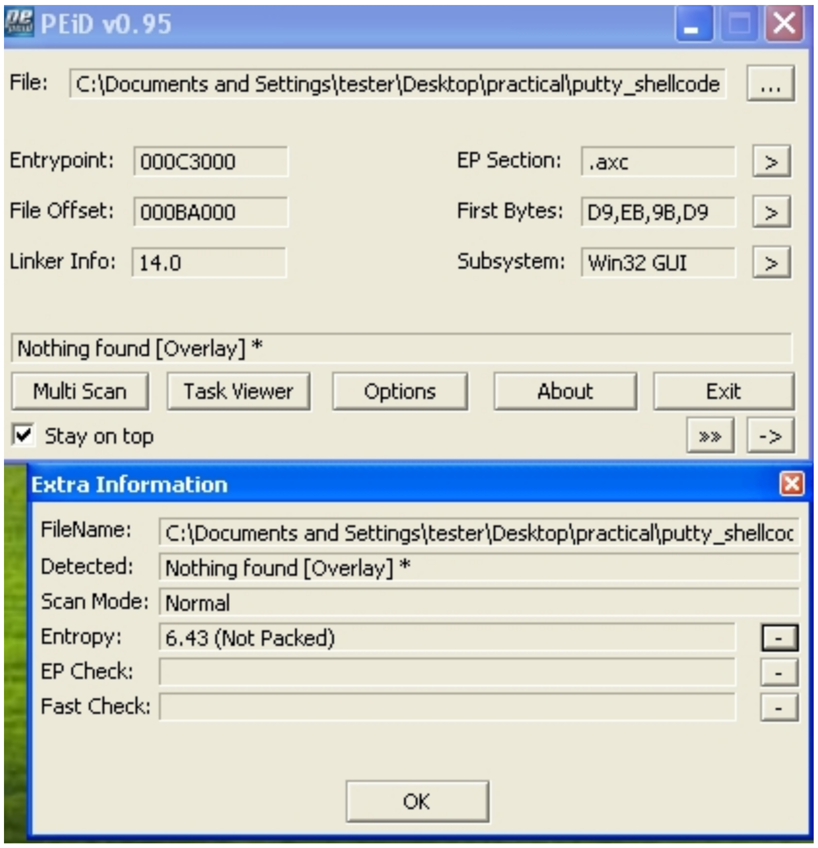
Here we can see that the addition of the new section with the shellcode actually causes a minor reduction in the overall entropy of the original PE file, instead of increasing it.



```
testbed@testbed:~/petests$ python section_entropy.py
putty_shellcode.exe
.00cfg
0.0611628522412
.rdata
6.00748644901
.bss
0.0
.data
2.72550188661
.gfids
1.96866876627
.rsrc
3.93375698954
.text
6.58712051468
.xdata
2.01874702565
.idata
5.56381456875
.reloc
6.7287146089
.axc
6.68389663176
```

Image 11: putty.exe - shellcode in section (section entropy)

The entropy of the section where the shellcode is stored has a value of 6.68 which also explains the minor reduction of the overall entropy value.



File: C:\Documents and Settings\tester\Desktop\practical\putty_shellcode

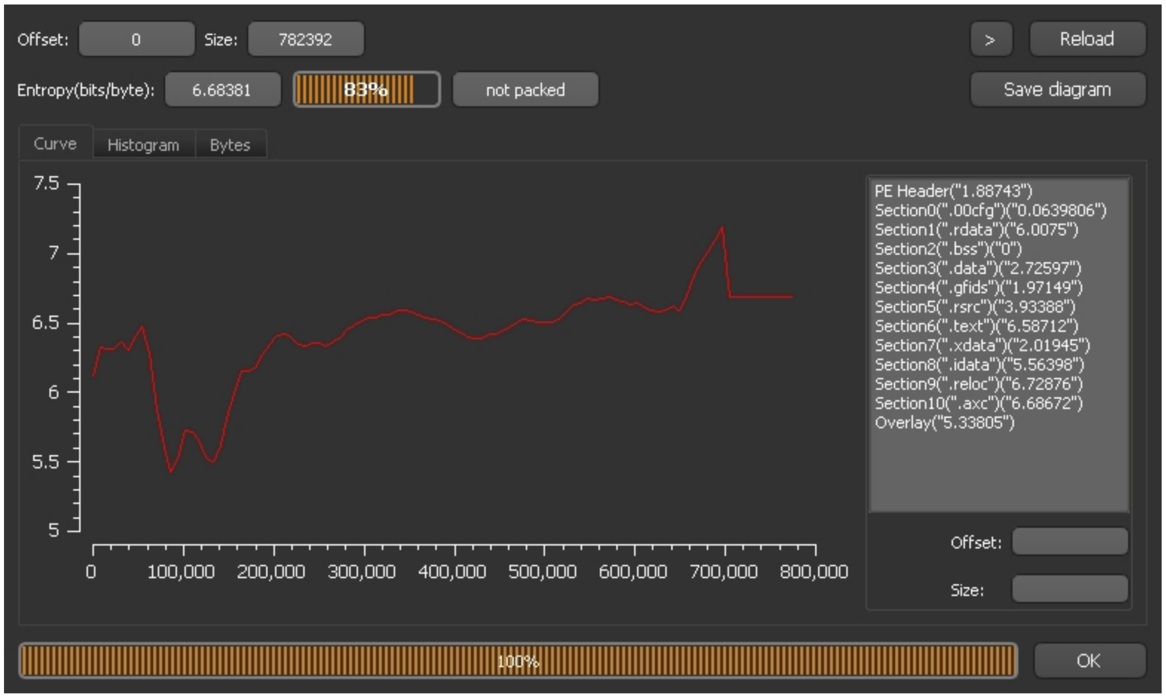
Entrypoint: 000C3000
EP Section: .axc
File Offset: 000BA000
First Bytes: D9,EB,9B,D9
Linker Info: 14.0
Subsystem: Win32 GUI

Nothing found [Overlay] *

FileName: C:\Documents and Settings\tester\Desktop\practical\putty_shellcoc
Detected: Nothing found [Overlay] *
Scan Mode: Normal
Entropy: 6.43 (Not Packed)
EP Check:
Fast Check:

Image 12: putty.exe - shellcode in section (PEiD entropy)

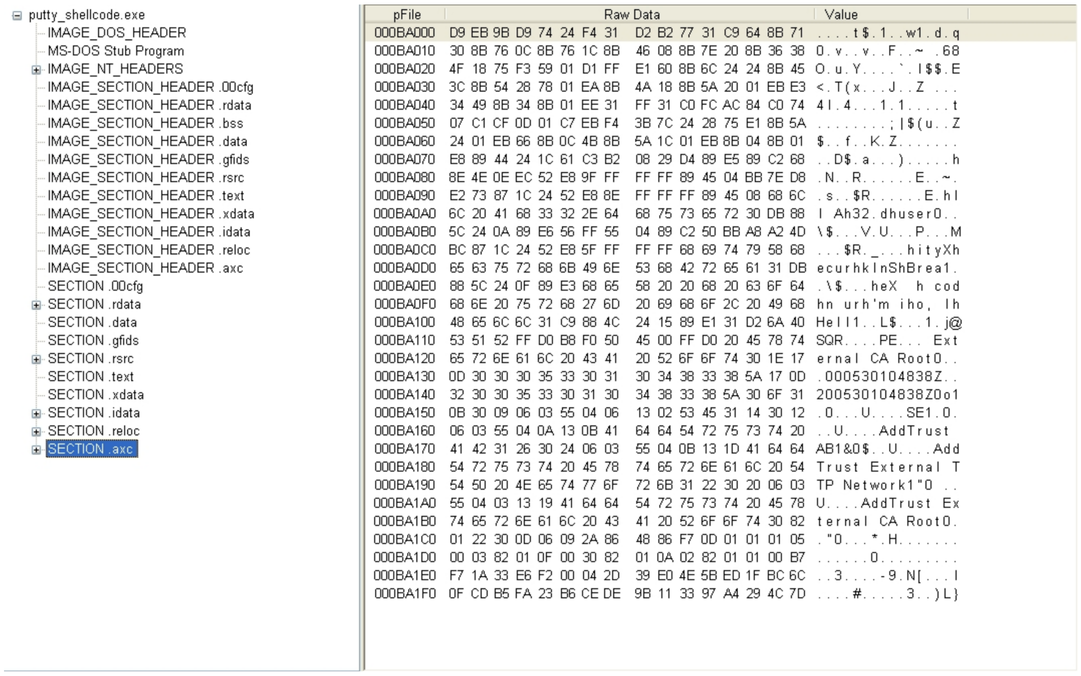
PEid does not report any changes to the entropy value which would considered normal since the shellcode itself is very small compared to the overall size of the executable.



```
PE Header("1.88743")
Section0(".00cfg")("0.0639806")
Section1(".rdata")("6.0075")
Section2(".bss")("0")
Section3(".data")("2.72597")
Section4(".gfids")("1.97149")
Section5(".rsrc")("3.93388")
Section6(".text")("6.58712")
Section7(".xdata")("2.01945")
Section8(".idata")("5.56398")
Section9(".reloc")("6.72876")
Section10(".axc")("6.68672")
Overlay("5.33805")
```

Image 13: putty.exe - shellcode in section (DIE entropy)

We can verify our findings about entropy from DIE as well both the overall and the entropy values of all the sections.



```
⊟ putty_shellcode.exe
    IMAGE_DOS_HEADER
    MS-DOS Stub Program
  ⊞ IMAGE_NT_HEADERS
    IMAGE_SECTION_HEADER .00cfg
    IMAGE_SECTION_HEADER .rdata
    IMAGE_SECTION_HEADER .bss
    IMAGE_SECTION_HEADER .data
    IMAGE_SECTION_HEADER .gfids
    IMAGE_SECTION_HEADER .rsrc
    IMAGE_SECTION_HEADER .text
    IMAGE_SECTION_HEADER .xdata
    IMAGE_SECTION_HEADER .idata
    IMAGE_SECTION_HEADER .reloc
    IMAGE_SECTION_HEADER .axc
    SECTION .00cfg
  ⊞ SECTION .rdata
    SECTION .data
    SECTION .gfids
  ⊞ SECTION .rsrc
    SECTION .text
    SECTION .xdata
  ⊞ SECTION .idata
  ⊞ SECTION .reloc
  ⊞ SECTION .axc
```

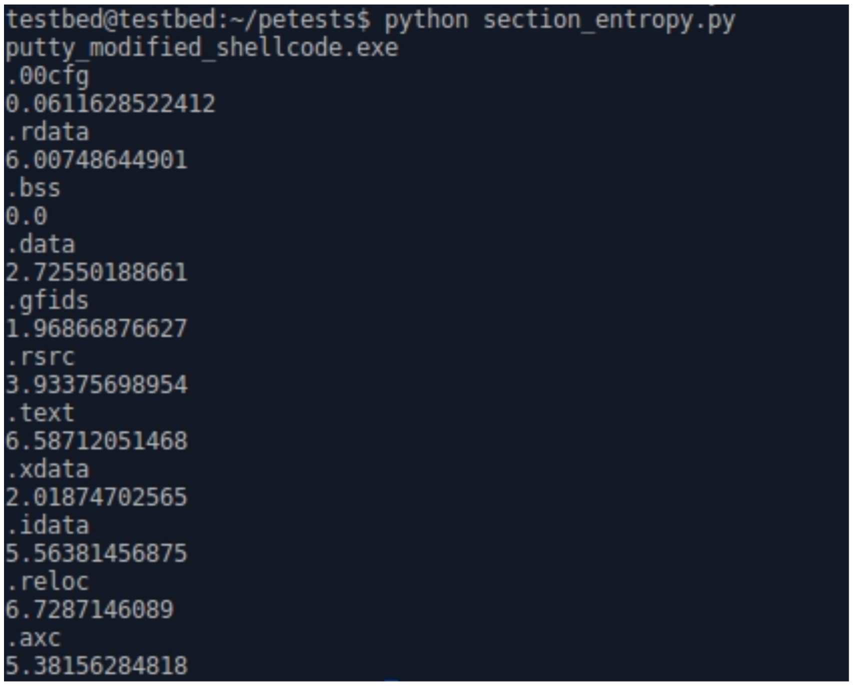
```
pFile     Raw Data                                          Value
000BA000  D9 EB 9B D9 74 24 F4 31  D2 B2 77 31 C9 64 8B 71  ....t$.1..w1.d.q
000BA010  30 8B 76 0C 8B 76 1C 8B  46 08 8B 7E 20 8B 36 38  0.v..v..F..~ .68
000BA020  4F 18 75 F3 59 01 D1 FF  E1 60 8B 6C 24 24 8B 45  O.u.Y....`.l$$.E
000BA030  3C 8B 54 28 78 01 EA 8B  4A 18 8B 5A 20 01 EB E3  <.T(x...J..Z ...
000BA040  34 49 8B 34 8B 01 EE 31  FF 31 C0 FC AC 84 C0 74  4I.4...1.1.....t
000BA050  07 C1 CF 0D 01 C7 EB F4  3B 7C 24 28 75 E1 8B 5A  ........;|$(u..Z
000BA060  24 01 EB 66 8B 0C 4B 8B  5A 1C 01 EB 8B 04 8B 01  $..f..K.Z.......
000BA070  E8 89 44 24 1C 61 C3 B2  08 29 D4 89 E5 89 C2 68  ..D$.a...).....h
000BA080  8E 4E 0E EC 52 E8 9F FF  FF FF 89 45 04 BB 7E D8  .N..R......E..~.
000BA090  E2 73 87 1C 24 52 E8 8E  FF FF FF 89 45 08 68 6C  .s..$R......E.hl
000BA0A0  6C 20 41 68 33 32 2E 64  68 75 73 65 72 30 DB 88  l Ah32.dhuser0..
000BA0B0  5C 24 0A 89 E6 56 FF 55  04 89 C2 50 BB A8 A2 4D  \$...V.U...P...M
000BA0C0  BC 87 1C 24 52 E8 5F FF  FF FF 68 69 74 79 58 68  ...$R._...hityXh
000BA0D0  65 63 75 72 68 6B 49 6E  53 68 42 72 65 61 31 DB  ecurhkInShBrea1.
000BA0E0  88 5C 24 0F 89 E3 68 65  58 20 20 68 20 63 6F 64  .\$...heX  h cod
000BA0F0  68 6E 20 75 72 68 27 6D  20 69 68 6F 2C 20 49 68  hn urh'm iho, Ih
000BA100  48 65 6C 6C 31 C9 88 4C  24 15 89 E1 31 D2 6A 40  Hell1..L$...1.j@
000BA110  53 51 52 FF D0 B8 F0 50  45 00 FF D0 20 45 78 74  SQR....PE... Ext
000BA120  65 72 6E 61 6C 20 43 41  20 52 6F 6F 74 30 1E 17  ernal CA Root0..
000BA130  0D 30 30 30 35 33 30 31  30 34 38 33 38 5A 17 0D  .000530104838Z..
000BA140  32 30 30 35 33 30 31 30  34 38 33 38 5A 30 6F 31  200530104838Z0o1
000BA150  0B 30 09 06 03 55 04 06  13 02 53 45 31 14 30 12  .0...U....SE1.0.
000BA160  06 03 55 04 0A 13 0B 41  64 64 54 72 75 73 74 20  ..U....AddTrust 
000BA170  41 42 31 26 30 24 06 03  55 04 0B 13 1D 41 64 64  AB1&0$..U....Add
000BA180  54 72 75 73 74 20 45 78  74 65 72 6E 61 6C 20 54  Trust External T
000BA190  54 50 20 4E 65 74 77 6F  72 6B 31 22 30 20 06 03  TP Network1"0 ..
000BA1A0  55 04 03 13 19 41 64 64  54 72 75 73 74 20 45 78  U....AddTrust Ex
000BA1B0  74 65 72 6E 61 6C 20 43  41 20 52 6F 6F 74 30 82  ternal CA Root0.
000BA1C0  01 22 30 0D 06 09 2A 86  48 86 F7 0D 01 01 01 05  ."0...*.H.......
000BA1D0  00 03 82 01 0F 00 30 82  01 0A 02 82 01 01 00 B7  ......0.........
000BA1E0  F7 1A 33 E6 F2 00 04 2D  39 E0 4E 5B ED 1F BC 6C  ..3....-9.N[...l
000BA1F0  0F CD B5 FA 23 B6 CE DE  9B 11 33 97 A4 29 4C 7D  ....#.....3..)L}
```

Image 14: putty.exe - shellcode in section (PEview sections)

From PEView we can see the actual size of the section as well as the contents

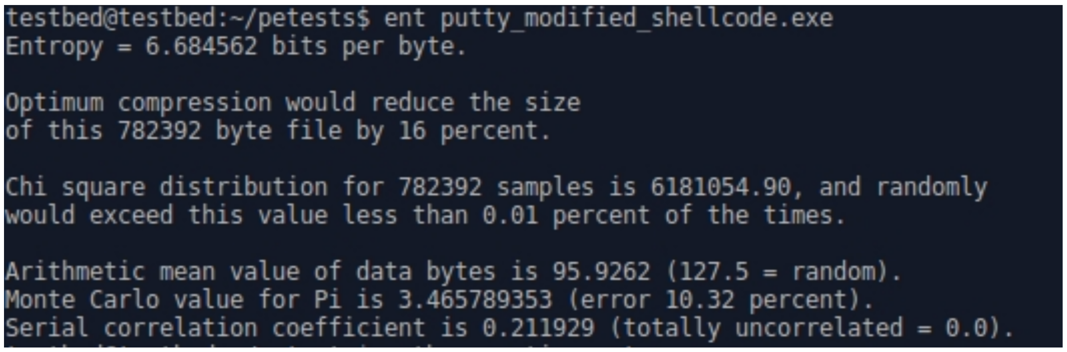
9.2.5 Adding mofidied shellcode to new section to test PE

In this step we are going to modify our shellcode to check if we can modify the section entropy that it is located. This is going to be accomplished by surrounding the shellcode with NOPs before and after.



```
testbed@testbed:~/petests$ python section_entropy.py
putty_modified_shellcode.exe
.00cfg
0.0611628522412
.rdata
6.00748644901
.bss
0.0
.data
2.72550188661
.gfids
1.96866876627
.rsrc
3.93375698954
.text
6.58712051468
.xdata
2.01874702565
.idata
5.56381456875
.reloc
6.7287146089
.axc
5.38156284818
```

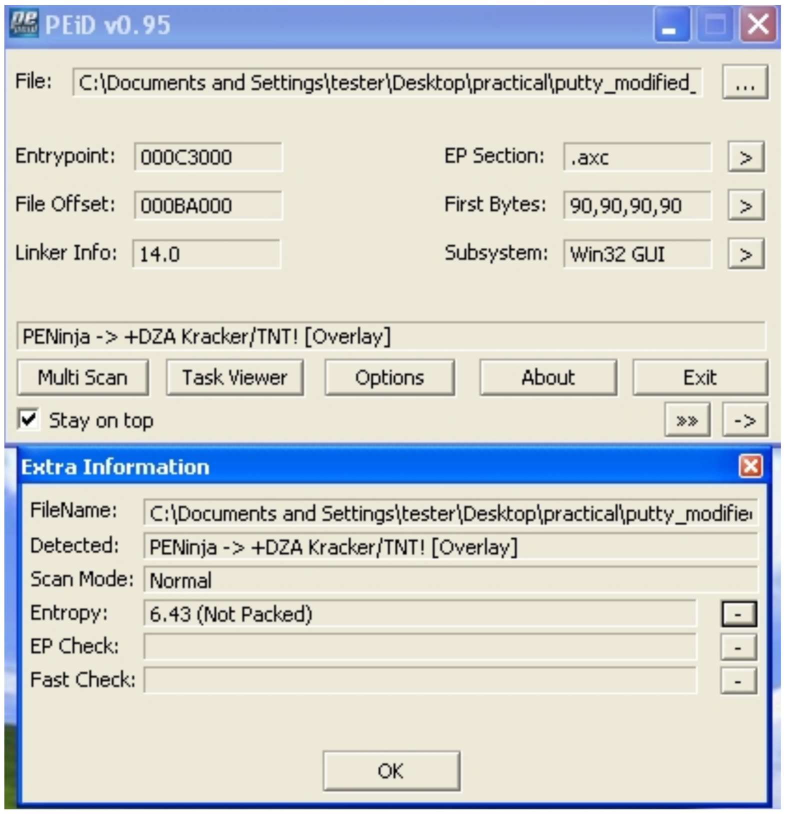
Image 15: putty.exe - modified shellcode in section (section entropy)



```
testbed@testbed:~/petests$ ent putty_modified_shellcode.exe
Entropy = 6.684562 bits per byte.

Optimum compression would reduce the size
of this 782392 byte file by 16 percent.

Chi square distribution for 782392 samples is 6181054.90, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 95.9262 (127.5 = random).
Monte Carlo value for Pi is 3.465789353 (error 10.32 percent).
Serial correlation coefficient is 0.211929 (totally uncorrelated = 0.0).
```

Image 16: putty.exe - modified shellcode in section (ent entropy)

By following the same procedure as the previous steps we gather the following results



PEiD v0.95

File: C:\Documents and Settings\tester\Desktop\practical\putty_modified_

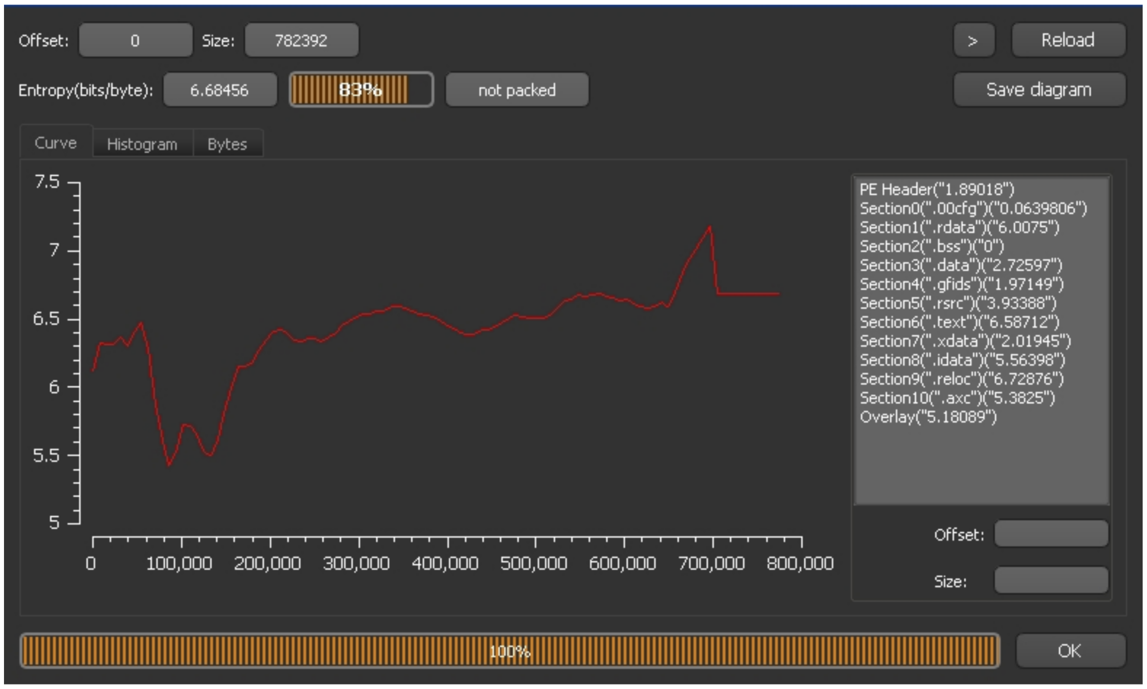
Entrypoint: 000C3000
EP Section: .axc
File Offset: 000BA000
First Bytes: 90,90,90,90
Linker Info: 14.0
Subsystem: Win32 GUI

PENinja -> +DZA Kracker/TNT! [Overlay]

Extra Information

FileName: C:\Documents and Settings\tester\Desktop\practical\putty_modifie
Detected: PENinja -> +DZA Kracker/TNT! [Overlay]
Scan Mode: Normal
Entropy: 6.43 (Not Packed)
EP Check:
Fast Check:

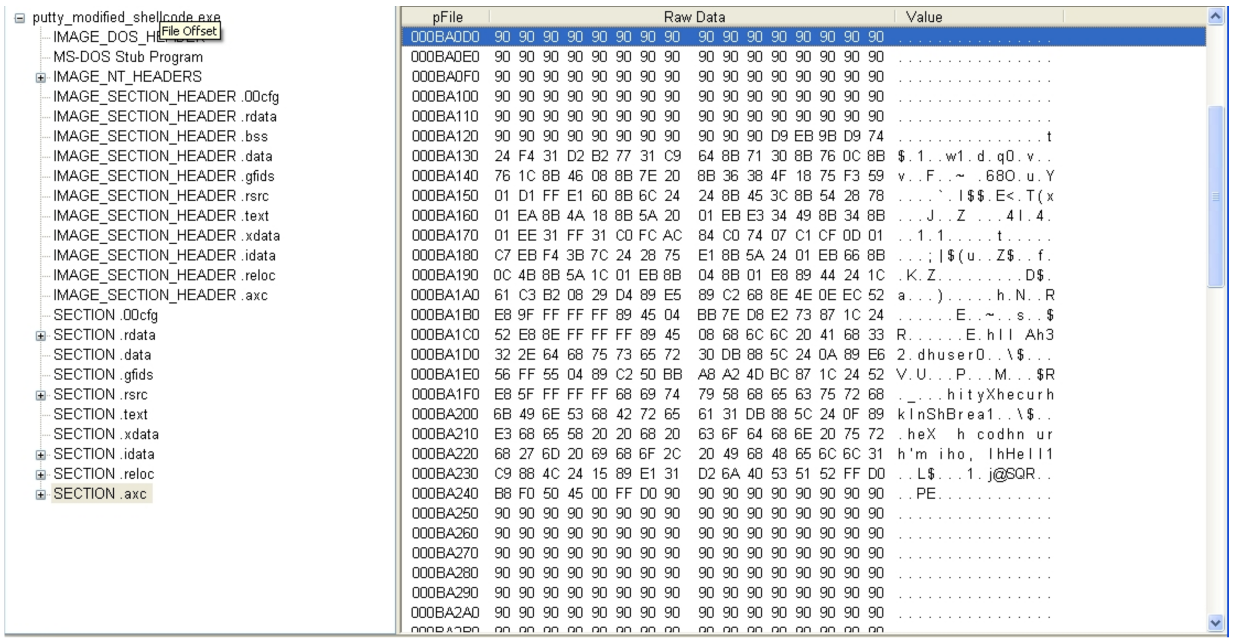
Image 17: putty.exe - modified shellcode in section (PEiD



Offset: 0 Size: 782392
Entropy(bits/byte): 6.68456 83% not packed

```
PE Header("1.89018")
Section0(".00cfg")("0.0639806")
Section1(".rdata")("6.0075")
Section2(".bss")("0")
Section3(".data")("2.72597")
Section4(".gfids")("1.97149")
Section5(".rsrc")("3.93388")
Section6(".text")("6.58712")
Section7(".xdata")("2.01945")
Section8(".idata")("5.56398")
Section9(".reloc")("6.72876")
Section10(".axc")("5.3825")
Overlay("5.18089")
```

Image 18: putty.exe - modified shellcode in section (DIE entropy)



```
putty_modified_shellcode.exe
  IMAGE_DOS_H  File Offset
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
  IMAGE_SECTION_HEADER .00cfg
  IMAGE_SECTION_HEADER .rdata
  IMAGE_SECTION_HEADER .bss
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .gfids
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .xdata
  IMAGE_SECTION_HEADER .idata
  IMAGE_SECTION_HEADER .reloc
  IMAGE_SECTION_HEADER .axc
  SECTION .00cfg
  SECTION .rdata
  SECTION .data
  SECTION .gfids
  SECTION .rsrc
  SECTION .text
  SECTION .xdata
  SECTION .idata
  SECTION .reloc
  SECTION .axc
```

```
pFile      Raw Data                                          Value
000BA0D0   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA0E0   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA0F0   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA100   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA110   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA120   90 90 90 90 90 90 90 90  90 90 90 D9 EB 9B D9 74  ...............t
000BA130   24 F4 31 D2 B2 77 31 C9  64 8B 71 30 8B 76 0C 8B  $.1..w1.d.q0.v..
000BA140   76 1C 8B 46 08 8B 7E 20  8B 36 38 4F 18 75 F3 59  v..F..~ .68O.u.Y
000BA150   01 D1 FF E1 60 8B 6C 24  24 8B 45 3C 8B 54 28 78  ....`.l$$.E<.T(x
000BA160   01 EA 8B 4A 18 8B 5A 20  01 EB E3 34 49 8B 34 8B  ...J..Z ...4I.4.
000BA170   01 EE 31 FF 31 C0 FC AC  84 C0 74 07 C1 CF 0D 01  ..1.1.....t.....
000BA180   C7 EB F4 3B 7C 24 28 75  E1 8B 5A 24 01 EB 66 8B  ...;|$(u..Z$..f.
000BA190   0C 4B 8B 5A 1C 01 EB 8B  04 8B 01 E8 89 44 24 1C  .K.Z.........D$.
000BA1A0   61 C3 B2 08 29 D4 89 E5  89 C2 68 8E 4E 0E EC 52  a...).....h.N..R
000BA1B0   E8 9F FF FF FF 89 45 04  BB 7E D8 E2 73 87 1C 24  ......E..~..s..$
000BA1C0   52 E8 8E FF FF FF 89 45  08 68 6C 6C 20 41 68 33  R......E.hll Ah3
000BA1D0   32 2E 64 68 75 73 65 72  30 DB 88 5C 24 0A 89 E6  2.dhuser0..\$...
000BA1E0   56 FF 55 04 89 C2 50 BB  A8 A2 4D BC 87 1C 24 52  V.U...P...M...$R
000BA1F0   E8 5F FF FF FF 68 69 74  79 58 68 65 63 75 72 68  ._...hityXhecurh
000BA200   6B 49 6E 53 68 42 72 65  61 31 DB 88 5C 24 0F 89  kInShBrea1..\$..
000BA210   E3 68 65 58 20 20 68 20  63 6F 64 68 6E 20 75 72  .heX  h codhn ur
000BA220   68 27 6D 20 69 68 6F 2C  20 49 68 48 65 6C 6C 31  h'm iho, IhHell1
000BA230   C9 88 4C 24 15 89 E1 31  D2 6A 40 53 51 52 FF D0  ..L$...1.j@SQR..
000BA240   B8 F0 50 45 00 FF D0 90  90 90 90 90 90 90 90 90  ..PE............
000BA250   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA260   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA270   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA280   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA290   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000BA2A0   90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
```

Image 19: putty.exe - modified shellcode in section (PEview sections)

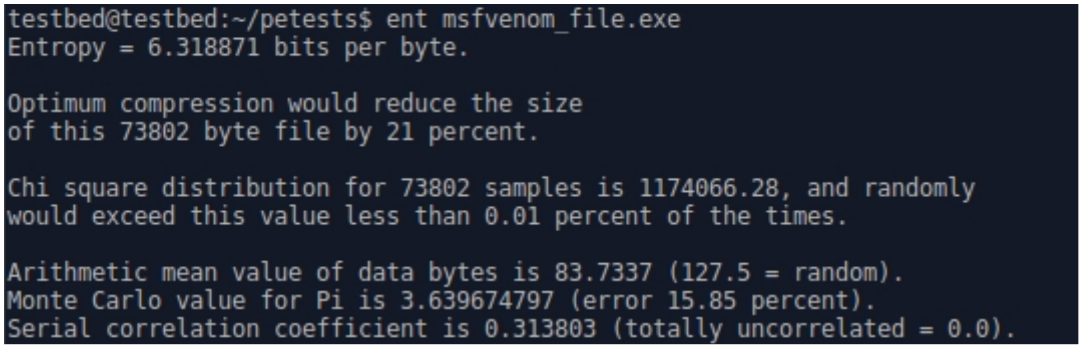
As we can see from the above screenshots we get the following results:

1. The overall entropy of the file reduced slightly but not as much as when we added only the shellcode. This will be explained later.
2. The section entropy where the shellcode is is now reduced to a value around 5.3, a difference of 1.3 less from when the shellcode was not surrounded with NOPs.
3. PEid reports that the overall entropy of the file was not reduced which again is expected if we compare the size of the section and the size of the whole file.
4. DIE reports values overall and section very similar to the values reported from ent and our python code.
5. In the final Screenshot we can see the shellcode surrounded with NOPs as expected. However because the shellcode size and the number of NOPs that we inserted was lower than the size of the section we created, it was observed the same issue as a previous example. For the bytes of the section that we didn't fill, they were filled with random values, thus raising the section entropy to 5.3. If we had filled the whole section with only the shellcode and NOPs the entropy value would be much lower that the one that we got. This is also the reason that when we used ent we got a slightly higher entropy value.

9.3 Creating PE from shellcode

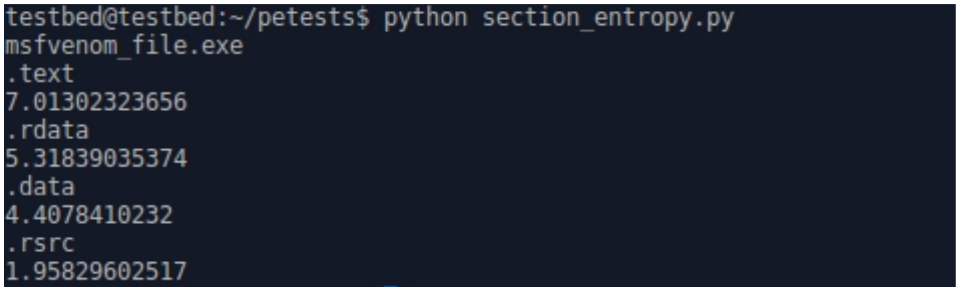
For this part we will generate an executable from msfvenom that will be encrypted. The shellcode that will be used will be a bind tcp shell shellcode encoded with shikata_ga_nai.

We test the results using the way we did for the previous parts



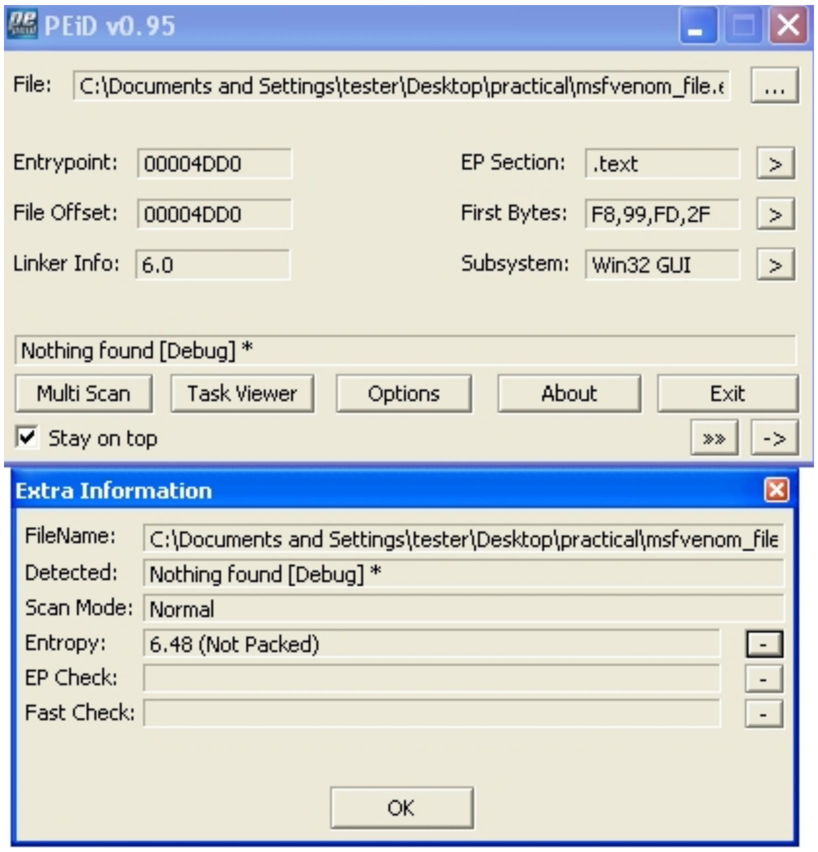
```
testbed@testbed:~/petests$ ent msfvenom_file.exe
Entropy = 6.318871 bits per byte.

Optimum compression would reduce the size
of this 73802 byte file by 21 percent.

Chi square distribution for 73802 samples is 1174066.28, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 83.7337 (127.5 = random).
Monte Carlo value for Pi is 3.639674797 (error 15.85 percent).
Serial correlation coefficient is 0.313803 (totally uncorrelated = 0.0).
```

Image 20: PE from shellcode (ent entropy)



```
testbed@testbed:~/petests$ python section_entropy.py
msfvenom_file.exe
.text
7.01302323656
.rdata
5.31839035374
.data
4.4078410232
.rsrc
1.95829602517
```

Image 21: PE from shellcode (section entropy)


PEiD v0.95

File: C:\Documents and Settings\tester\Desktop\practical\msfvenom_file.e

Entrypoint: 00004DD0 EP Section: .text

File Offset: 00004DD0 First Bytes: F8,99,FD,2F

Linker Info: 6.0 Subsystem: Win32 GUI

Nothing found [Debug] *

☑ Stay on top

Extra Information

FileName: C:\Documents and Settings\tester\Desktop\practical\msfvenom_file

Detected: Nothing found [Debug] *

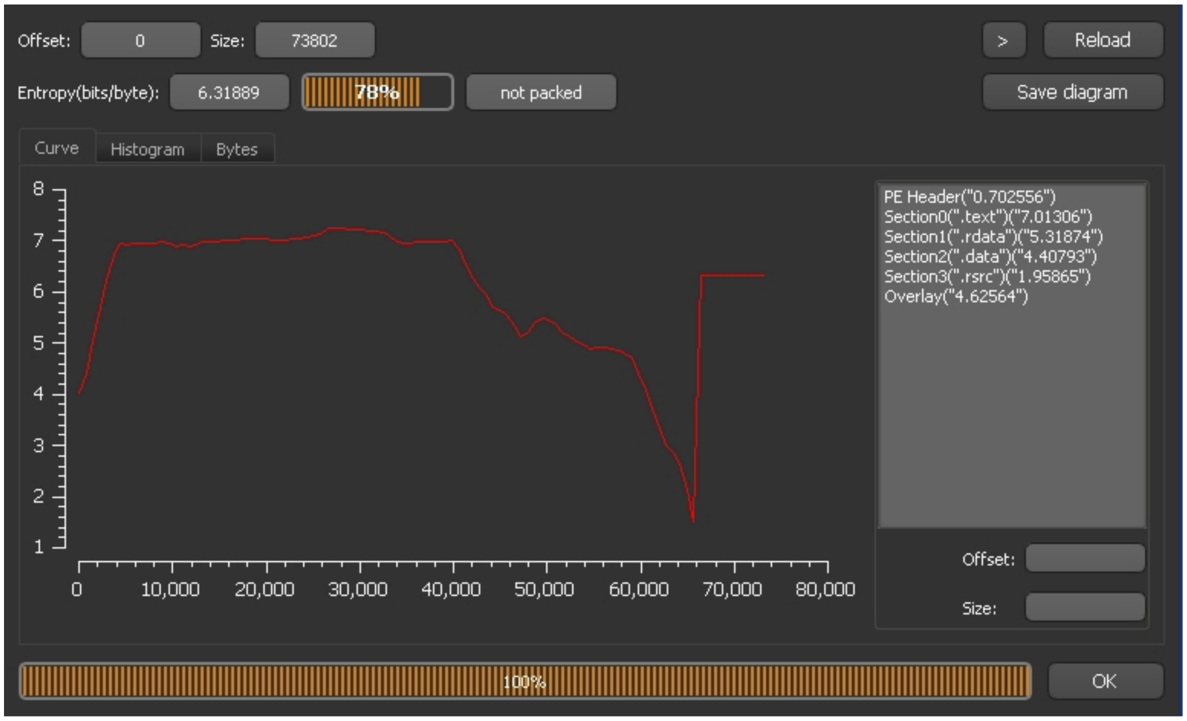
Scan Mode: Normal

Entropy: 6.48 (Not Packed)

EP Check:

Fast Check:

Image 22: PE from shellcode (PEiD entropy)


Offset: 0 Size: 73802

Entropy(bits/byte): 6.31889 78% not packed

PE Header("0.702556")
Section0(".text")("7.01306")
Section1(".rdata")("5.31874")
Section2(".data")("4.40793")
Section3(".rsrc")("1.95865")
Overlay("4.62564")

Image 23: PE from shellcode (DIE entropy)

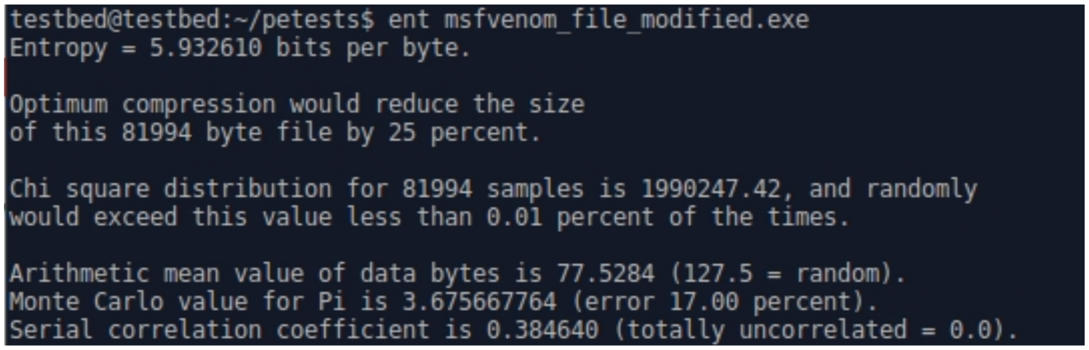
The results we get are the following:

- The overall entropy of the PE is high at 6.31 using ent and DIE and 6.48 using PEid
- The high value is expected as the the shellcode has been encrypted

9.3.1 Adding section to generated PE

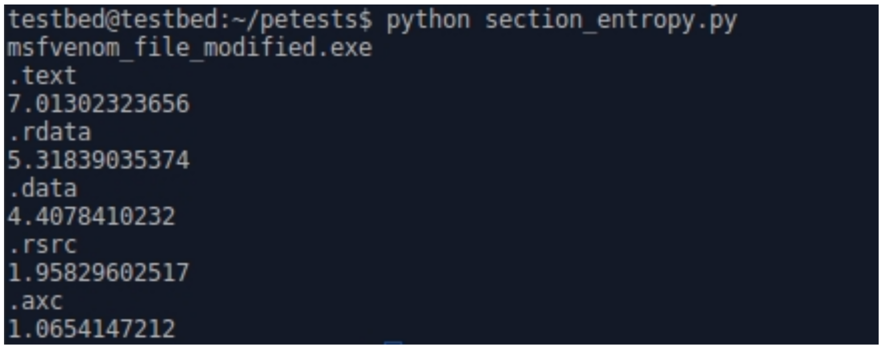
In this part we will use the PE that we generated from msfvenom and we will add to in a section that will be filled with NOPs to test the effect on the overall entropy of the file.

The check the results using the same method.



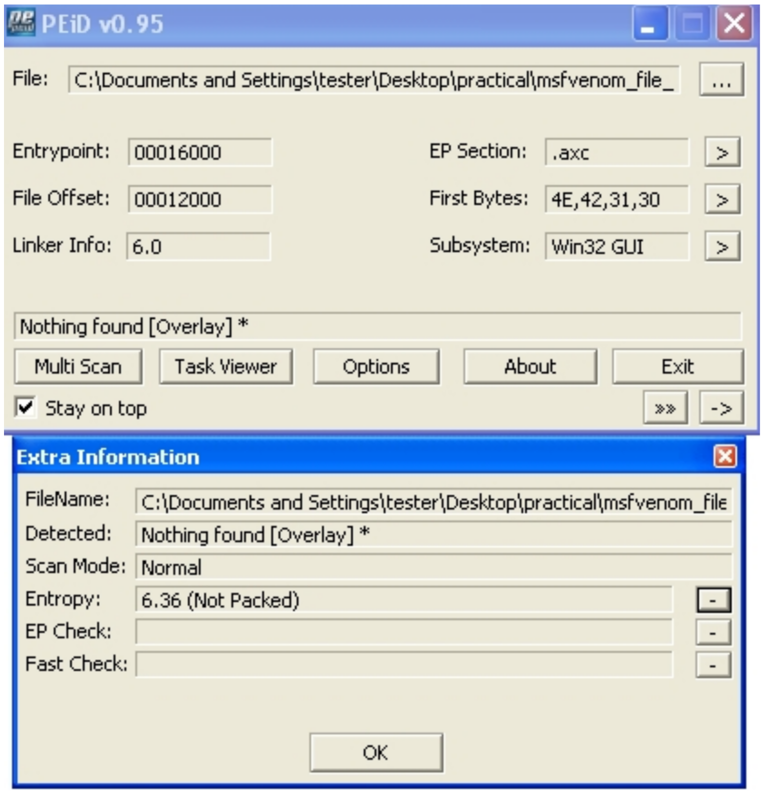
```
testbed@testbed:~/petests$ ent msfvenom_file_modified.exe
Entropy = 5.932610 bits per byte.

Optimum compression would reduce the size
of this 81994 byte file by 25 percent.

Chi square distribution for 81994 samples is 1990247.42, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 77.5284 (127.5 = random).
Monte Carlo value for Pi is 3.675667764 (error 17.00 percent).
Serial correlation coefficient is 0.384640 (totally uncorrelated = 0.0).
```

Image 24: PE from shellcode - modified (ent entropy)



```
testbed@testbed:~/petests$ python section_entropy.py
msfvenom_file_modified.exe
.text
7.01302323656
.rdata
5.31839035374
.data
4.4078410232
.rsrc
1.95829602517
.axc
1.0654147212
```

Image 25: PE from shellcode - modified (section entropy)



PEiD v0.95

File: C:\Documents and Settings\tester\Desktop\practical\msfvenom_file_

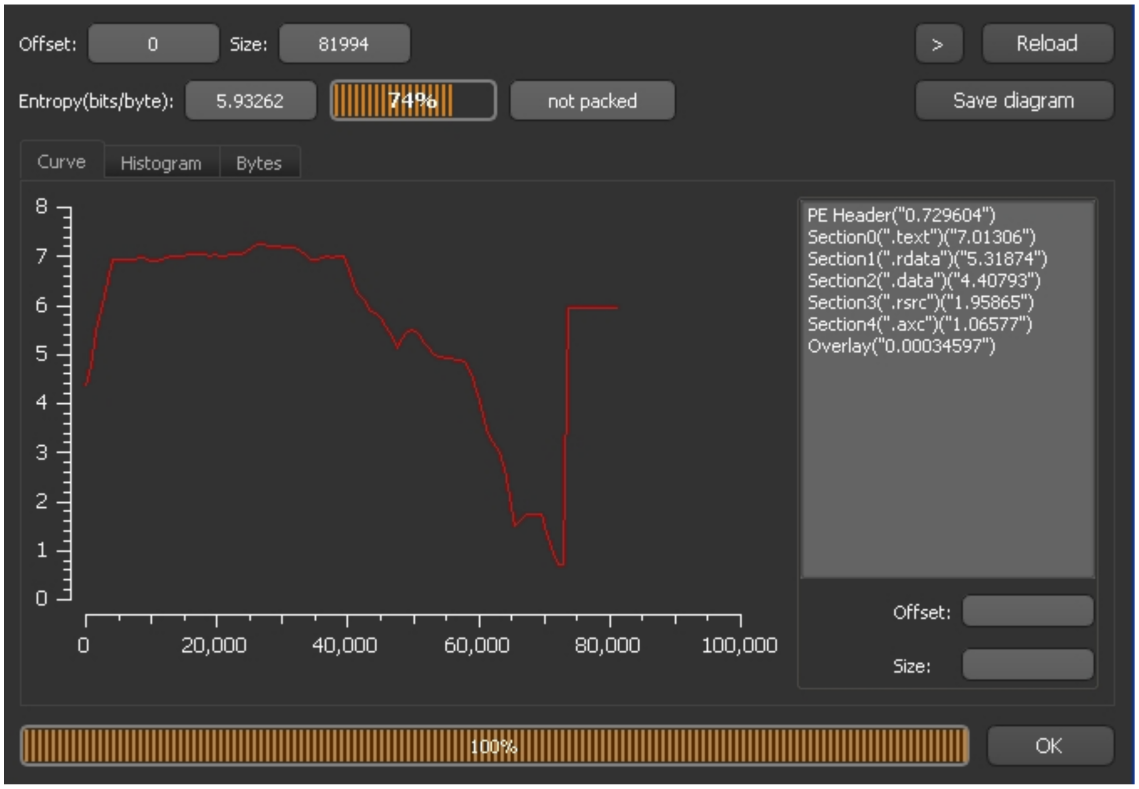
Entrypoint: 00016000 | EP Section: .axc
File Offset: 00012000 | First Bytes: 4E,42,31,30
Linker Info: 6.0 | Subsystem: Win32 GUI

Nothing found [Overlay] *

Extra Information

FileName: C:\Documents and Settings\tester\Desktop\practical\msfvenom_file
Detected: Nothing found [Overlay] *
Scan Mode: Normal
Entropy: 6.36 (Not Packed)
EP Check:
Fast Check:

Image 26: PE from shellcode - modified (PEiD entropy)

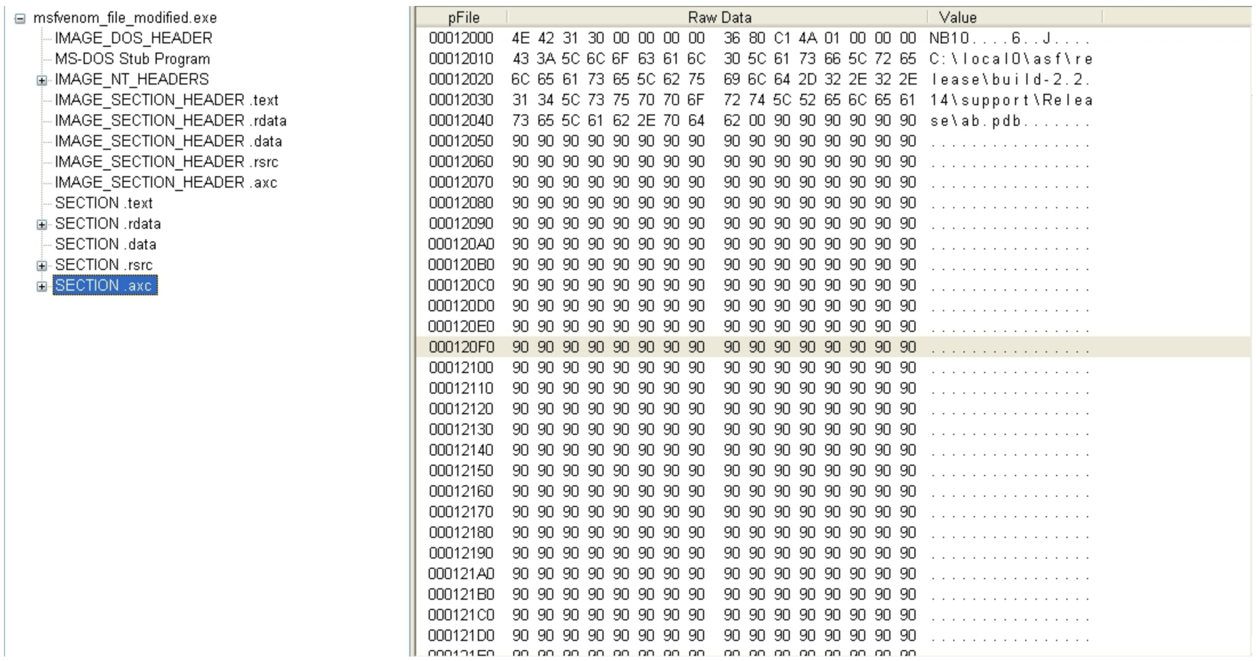


Offset: 0 Size: 81994

Entropy(bits/byte): 5.93262 74% not packed

```
PE Header("0.729604")
Section0(".text")("7.01306")
Section1(".rdata")("5.31874")
Section2(".data")("4.40793")
Section3(".rsrc")("1.95865")
Section4(".axc")("1.06577")
Overlay("0.00034597")
```

Image 27: PE from shellcode - modified (DIE entropy)



```
msfvenom_file_modified.exe
    IMAGE_DOS_HEADER
    MS-DOS Stub Program
  + IMAGE_NT_HEADERS
    IMAGE_SECTION_HEADER .text
    IMAGE_SECTION_HEADER .rdata
    IMAGE_SECTION_HEADER .data
    IMAGE_SECTION_HEADER .rsrc
    IMAGE_SECTION_HEADER .axc
    SECTION .text
  + SECTION .rdata
    SECTION .data
  + SECTION .rsrc
  + SECTION .axc
```

```
pFile     Raw Data                                          Value
00012000  4E 42 31 30 00 00 00 00  36 80 C1 4A 01 00 00 00  NB10....6..J....
00012010  43 3A 5C 6C 6F 63 61 6C  30 5C 61 73 66 5C 72 65  C:\local0\asf\re
00012020  6C 65 61 73 65 5C 62 75  69 6C 64 2D 32 2E 32 2E  lease\build-2.2.
00012030  31 34 5C 73 75 70 70 6F  72 74 5C 52 65 6C 65 61  14\support\Relea
00012040  73 65 5C 61 62 2E 70 64  62 00 90 90 90 90 90 90  se\ab.pdb.......
00012050  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012060  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012070  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012080  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012090  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000120A0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000120B0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000120C0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000120D0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000120E0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000120F0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012100  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012110  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012120  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012130  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012140  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012150  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012160  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012170  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012180  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00012190  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000121A0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000121B0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000121C0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
000121D0  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
```

Image 28: PE from shellcode - modified (PEview sections)

As we can see the results are what we would expect:

The Overall entropy using all the tools (ent, PEid, DIE) has been lowered. Not by a significant amount since the size of the PE is 72KB and the section that we are creating is around 1KB. Also something else that we need to take into account is that the section is not full with NOP instructions.

9.4 Summary Table

The following table represents a summary of the findings for the files that we tested, focusing on the overall entropy and on the entropy of the new section we created (.axc)

File Type	Measure Method					File Name
	ent	Peid	DIE	DIE (.axc)	python(.axc)	
Original Test PE	6.72646	6.43	6.72646	N/A	N/A	putty.exe
Added NOP Section	6.685167	6.42	6.68517	1.4729	1.4719	putty_modified.exe
Added Shellcode section	6.683808	6.43	6.68381	6.68672	6.6838	putty_shellcode.exe
Added modified shellcode	6.684562	6.43	6.68456	5.3825	5.3815	putty_modified_shellcode.exe
Shellcode Executable	6.318871	6.48	6.31889	N/A	N/A	msfvenom_file.exe
Modified shellcode	5.93261	6.36	5.93262	1.06577	1.06541	msfvenom_file_modified.exe

9.5 Entropy of Malware Samples

As part of the testings a number of malware samples available online on different repositories were tested for their entropy values.

The Following table represents a small sample of the 3700 files that were tested.

The Collumn “AVG_ENTROPY” refers to the overall entropy of the file while the “MIN” and “MAX” entropy collumns refer to the minumum and maximum entropy values of the sections

AVG_ENTROPY	MIN_ENTROPY	MAX_ENTROPY
4.426297303	2.867124198	6.096939721
5.214709787	4.084872622	6.673501782
3.173186073	0	7.999371708
2.294760411	0	6.530868468
2.294783725	0	6.530868468
2.327657194	0	6.631747641
6.017597431	3.402931054	7.902031708
2.300557835	0	6.562232258
2.295011308	0	6.530868468
2.297530702	0	6.564721241
4.479505412	0	6.449717905
2.305030692	0	6.562232258
2.294803859	0	6.530868468
6.227480168	4.184278832	7.947140697
5.99050511	4.619834149	7.997200275
3.925425342	0	7.790831956
3.973397216	1.892199801	6.516178183
5.382012912	4.445088033	6.638992819
3.733659266	0	6.497884652
4.304746956	2.399821812	6.567290989
2.294654207	0	6.530868468
2.295313201	0	6.530868468
2.294979921	0	6.530868468
2.295129445	0	6.530868468
4.599600543	2.988368347	6.577777545
2.295002922	0	6.530868468
4.165283174	0.269444839	7.998463401
4.02028889	0	6.396305005
3.426560412	0	6.650422302
4.441162392	0.394140975	6.683800888
4.248977944	0.60747647	6.683642715
4.482634099	2.94889919	6.550136664
2.294939734	0	6.530868468
2.298667213	0	6.52973184
2.345772926	0	6.499603846
6.030917672	5.10382227	6.89497467
4.577271241	0	6.598411597
2.295159016	0	6.530868468
2.330158428	0	6.496038061
5.245474628	1.836679167	7.998993291
2.296986435	0	6.53892752
2.294639202	0	6.530868468
4.137655425	0	6.410406825
4.955723175	3.667153173	6.419912706
2.433021436	0	6.563139352
4.441406182	2.927559713	6.096939721
2.297264108	0	6.53892752
2.29483193	0	6.530868468
2.29502051	0	6.530868468
5.958414586	4.591708551	7.874263165
2.294577923	0	6.530868468
2.294920758	0	6.530868468
2.294671195	0	6.530868468
3.454583804	0	7.93600008
2.760652352	0	5.910445078
2.915807825	0	5.935052364
2.258390325	0	5.860265633
4.9307686	3.193638902	6.605107474
6.597119433	3.985078565	7.983563618
4.982680109	4.127240712	6.32622336
4.356458536	0	6.417698237
4.98598922	2.951925917	6.446828762
3.623639539	0	6.497884652
3.365807708	0	7.995181989
5.334365373	4.068465151	6.035730827
4.115510688	0	6.877406246
4.163910847	0	6.497884652
5.287804455	4.207579475	6.572551931
4.917674437	2.458886248	6.318474099
5.586331992	4.508373407	6.561306487
4.821959119	3.202715463	6.516518185
2.544826765	0	5.974830461
2.329866222	0	6.153356413
5.179280157	4.303255992	6.573206125
4.467429119	0	7.927776502
4.316136169	0	7.926069174
4.038051772	0	7.917869782
4.153434456	0	7.93152154
4.441803949	0	7.780490017
5.097277607	2.43668428	6.543951938
3.41737148	0	6.497884652

AVG_ENTROPY	MIN_ENTROPY	MAX_ENTROPY	AVG_ENTROPY	MIN_ENTROPY	MAX_ENTROPY
4.610731649	2.152015628	6.68581953	4.517077282	0.020393135	6.599033393
3.941302849	0	7.998987038	4.873116266	3.192076594	6.682816882
4.026921653	1.852499945	6.464695388	3.18619056	0	6.015548456
2.649535825	0	5.589445261	3.961854917	0	6.717667374
5.049897405	3.89961076	6.642460579	2.938541563	0	5.904659367
2.328679917	0	5.862227917	2.244044716	0	5.809670488
4.138339841	0	6.417698237	3.740301655	0	6.497884652
4.47585472	0	7.776313108	3.408559137	0	7.81835259
5.034868575	2.488179651	6.394668199	4.421767704	1.955565864	6.528199949
3.321002224	0	5.997496784	4.889881091	4.080367057	6.614560896
3.361344469	0	6.535749259	3.55705392	0	6.497884652
5.35824472	4.396796299	6.579148586	3.847562126	0	6.497884652
3.967695937	0	6.497884652	4.683639081	3.189745702	6.593136124
5.289034099	2.213200384	7.05695537	3.120867614	0	6.497884652
2.673733682	0	6.436379637	4.494347293	2.808535214	6.387448572
2.473338857	0	6.497884652	3.590930721	0	7.474678754
3.302447264	0	6.497884652	4.43589276	0	7.924875779
3.258094291	0	6.480448376	3.513839978	0	6.497884652
4.884486487	2.266386129	6.578698604	5.109330642	0	7.928653506
3.314309123	0	7.998374087	3.805065211	0	6.696397554
4.205485806	0	7.927312134	4.902661186	3.190142906	6.555858682
4.972576007	3.22351651	6.589233028	3.834712346	0	6.128108318
3.590692802	0	6.497884652	2.520830449	0	6.065012874
4.265277681	0	7.920170245	3.933574671	0.762670683	6.257227514
4.395258429	0	6.450231726	3.046336629	0	5.313254611
4.280595634	0	6.29093337	3.905956185	0	6.497884652
3.801349375	0	6.497884652	3.623504828	0	7.266887287
2.855868199	0	6.497884652	4.131767069	0	7.997699313
3.040766301	0	6.497884652	4.035109547	0	6.497884652
4.06877138	0	6.719415702	2.449333951	0	6.062043662
4.469655097	0	7.901445667	4.468556311	0	7.926680234
3.521821281	0	7.918980839	4.012265429	0.020393135	6.581870645
2.244044716	0	5.809670488	2.48198126	0	6.497884652
3.560271109	0	6.497884652	4.614002314	0.020393135	6.856402239
4.510238829	0	7.942032612	4.526147426	0	7.035102788
3.440589843	0	6.421092436	4.238204985	0.020393135	6.774073328
3.942427942	3.084965637	5.949718001	4.789055745	3.205876702	6.665481102
4.194853593	0	6.433100348	5.570017255	3.327596729	6.602102655
4.042752619	0	6.497884652	4.8211787	3.197740598	6.636600349
4.952370832	3.51044725	6.655914168	5.170584106	4.123284643	6.681036423
2.949007111	0	5.681021902	3.786191938	0	6.497884652

As we can see from the results the overall entropy and the section entropy of the malware samples varies. We would expect that the malware files would usually have high entropy values because of the encryption and the packaging but the numbers show something very different.

The values range from as low as 2.5 until as high as 6.5 with many files to have an average of 4.5 till 5.5. This is considered in many cases a value that can be found In legitimate software as well.

That makes us further question how accurate is the entropy value of a file to identify a potential malicious file, when many of the malwares tested that are available online have reasonable entropy values that can also pass as legitimate.

10 Final Results

From the tests that we did and their results, we come to an interesting conclusion. Even though malware authors choose to encrypt their files resulting to an increased entropy, the values can be modified easily by using the same methodology in this thesis, by adding new sections or surrounding the shellcode with a length of the same bytes to lower the entropy.

Anti-virus vendors use the entropy value as an indicator among others to identify potential malicious files.

By looking at the results we can see that the entropy cannot be safely used for the identification of malicious files, but it can still be used to identify potentially packed or encrypted files.

However not only malicious files are packed or encrypted. Many legitimate softwares have these traits. For example software installers have packed in one executable all the necessary files and dependencies for the software to be installed and run. Software authors use encryption techniques in their software to prevent reverse engineering to keep the code hard to read, but also many legitimate executables have a high entropy value just because they are big and have a big number of different and randomized characters in their code.

11 Future Work

As a future work on this thesis the tests performed can be expanded to other Operating Systems (OS). Specifically Android, because of the popularity of it and the fact that a very big percentage of current mobile devices have it pre-installed.

That reason has attracted the attention of malware authors, since the popularity of Android on devices also brought in the ecosystem users who are not aware of the dangers of now mobile malware, and not properly educated on the “on-line safety”.

Anti-virus vendors have released various solutions for protecting these devices by applying similar login as the other Operating Systems, therefore we can assume that the tests done in this thesis also apply in the case of android.

More specifically, in the case of encrypted and packed malware the same logic will apply in android as well. Creating test cases that show how can entropy can manipulated in the mobile ecosystem will show how much value it has as a metric to identify potential malicious application.

12 References

[1] “An HMM and structural entropy based detector for Android malware: An empirical study” - Geradro Canfora, Francesco Mercaldo, Corrado Aaron Visaggio

[2] “Comparing files using structural entropy” - Sorokin I

[3] “Structural entropy and metamorphic malware” - Baysa D

[4] “Information Theory and the Digital Age” - Aftab, Cheum, Kim, Thakkar, Yeddanapudi

[5] “The mathematical Theory of communication” - Shannon, Claude E, 1949

[6] “Entropy and Information Theory” - Robert M. Gray

[7] “Practical Malware Analysis” - Honig Andrew, Sikorski Michael, 2016

[8] “Hunting for metamorphic engines” - Wong W., Stamp M., 2006

[9] “Microsoft Portable Executable and Common Object File Format Specification”

[10] https://www.aldeid.com/wiki/PEiD

[11] http://ntinfo.biz/index.html

[12] http://wjradburn.com/software/PEview

[13] https://github.com/lsauer/entropy

[14] “Exploring Hidden Markov models for varus Analysis: A semantic Approach” - T.Austin, E. Filiol, S. Josse, M. Stamp (2013)

[15] “Computer Viruses and Malware” - J. Aycock , 2006

[16] “Difference between a computer virus and a computer worm” - http://scienceline.ucsb.edu/getkey.php?key=52

[17] “Classification of Malware Using Structured Control Flow” - S. Cesare and Y. Xiang (2010)

“Pattern Recognition and Machine Learning” - C. Bishop. (2006)

[18] “A Survey of Malware Detection Techniques” - N. Idika and A. Mathur (2007)

[19] “A Short Introduction to Boosting” - Y. Freund and R. Schapire (1999)

[20] “Learning to Detect and Classify Malicious Executables in the Wild” - S. Kolter and M. Maloof (2006).

[21] “Hidden Markov Models for Malware Classification” – Chinmayee Annachhatre (2013)

[22] “Panda Security (n.d.). Virus, worms, trojans and backdoors: Other harmful relatives of viruses” - http://www.pandasecurity.com/homeusers-cms3/security-info/about-malware/generalconcept

[23] “A tutorial on hidden Markov models and selected applications in speech recognition” - L. Rabiner (1989).

[24] “https://msdn.microsoft.com/library/windows/desktop/ms680547(v=vs.85).aspx”

[25] “https://technet.microsoft.com/en-us/library/dd632948.aspx”

[26] “Boot sector virus repair” - M. Landesman

[27] “What is the difference between viruses, worms, and Trojans?” - Symantec (2009)

[28] “Trojan Horse” - Symantec (2004)

[29] “Information Security: Principles and Practice , second edition” - M. Stamp (2011)

[30] “Metamorphic Detection via Emulation, Masters Thesis, San Jose State University” - S. Priyadarshani (2011)

[31] “Pattern Recognition and Machine Learning” - C. Bishop. (2006)

[32] “Toward an Automatic, Online Behavioral Malware Classifiction System” - R. Canzanese, M. Kam, and S. Mancoridis

[33] “Support-Vector Networks. Machine Learning” - C. Cortes and V. Vapnik(1995)