Detection of Heap-Based Overflow in binary Codes

by

Huma Siddiquie

A thesis submitted to the faculty of Information Security Department, Military College of Signals, National University of Sciences and Technology, Rawalpindi in partial fulfilment of the requirements for the degree of MS in Information Security

August 2018

Abstract

According to SANS Common weakness Enumeration Heap Overflow vulnerability is among twenty five most dangerous software errors which if exploited in an organized manner aids the attacker to gain privilege escalation. Detection of malwares that can exploit this vulnerability requires the combination of datamining and machine learning techniques.

Our work presents a hybrid malware detection technique that is the combination of both data mining and machine learning approach. For overcoming the absence of typical anti-virus software we have used static analysis technique to extract features of malwares. We extracted features from malware binaries then calling frequencies of the raw features are collected to select valuable features. Feature engineering technique is used for the reduction of the selected features. The created feature set is used to train three classifiers J48, K-Star and Simple logistic for the detection of malwares that exploit heap based overflow vulnerability. By embracing the notion of machine learning and datamining a static malware detection technique is proposed. The proposed technique is easy to implement in operations of cyber security to comprehend the behavior of malwares targeting their organizations.

Declaration

I hereby declare that no portion of work presented in this thesis has been submitted in support of another award or qualification either at this institution or elsewhere.

Dedication

“In the name of Allah, the most Beneficent, the most Merciful who bestow me with knowledge and favors me with opportunities to prove myself"

I dedicate this thesis to my parents, siblings, and teachers who had always been so encouraging and loving. Who always show their full confidence in me and make me to believe in myself.

Acknowledgments

I am thankful to ALMIGHTY ALLAH, the most Powerful and Gracious, Who helped me and guide me to successfully complete this thesis.

I wish to express my sincere thanks to my supervisor, Major. Muhammad Faisal Amjad, PhD, for his expert and valuable guidance, suggestions and ¬¬humbleness. Also, I would thank my committee members; Lecturer Waleed Bin Shahid and Asstt Prof. Mian Muhammad Waseem Iqbal for their support and knowledge regarding this topic.

Last, but not the least, I am highly thankful to my parents. They have always stood by my dreams and aspirations and have been a great source of inspiration for me. I would like to thank Sir Omer and Madam Anum for helping me in my tough time and this dissertation would have not been possible without their help.

Table of Contents

Page

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

A heap overflow is a well-known type of buffer overflow occurring in heap data area. Memory on heap usually comprises of program data and is dynamically allocated by the application at run-time. Heap is a segment of memory that is used for keeping global variables and dynamically allocated data. It can be exploited by corrupting the program data located at heap. It usually happens when a pointer or its index is step-down to a position before the buffer or when a destructive index is used, which produces a position before the buffer. Each portion of memory in heap comprises of boundary tags that enclose information related to memory management [1].

When a heap buffer is over run, the control statistics in these tags is overwritten. Access violation occurs when a memory address overwrite takes place. When the overflow is executed in an organized manner, the vulnerability would permit an attacker to overwrite a memory location with a carved input.

This vulnerability directly affects the CIA triad and can result in different consequences as specified below:

a. **Confidentiality.** By exploiting CWE-122 the attacker can read memory, execute unauthorized programs and can evade protection mechanism.
b. **Integrity.** It is mostly used by attackers to run arbitrary or unauthorized programs to modify memory. The memory modification can be done by overwriting function pointer residing in memory or pointing it to the exploiter's code
c. **Availability.** It can be used to crash the systems, generated D-Dos Attacks, resource consumption and can lead to put the program in to an infinite loop.

Several techniques have been recommended to identify this software vulnerability, but all of them rely on deep code analysis and run time execution which is a tedious process. Data mining and machine learning techniques have introduced new dimensions in the field of malware analysis. We have a proposed a hybrid detection technique that is the combination of both data mining and machine learning. Machine Learning technique is used to find patterns in data and then prediction of the outcome is done by using datamining technique [2].

We extracted twenty one features from malware executables which are File name, Risk Parameter, Network Connections, Number of Mutex, Number of Loaded libraries, Number of Process Interactions, DNS Queries, Frequent API Calls, Downloaded Files, Process Interactions, Registry Writes, Registry Reads, Mutex Count, Mutex Name, File Queries, Type of Strings, Total Count of Strings, Strings, Total Count Loaded Libraries (.dll format), Loaded Libraries (.dll format) and Type of Malware by self-written code in php language. Our work presents a heap based over flow vulnerability detection system using data mining technique such as three classifiers: J48, KStar and Simple Logistic. For overcoming the absence of typical anti-virus software we have used static analysis technique to extract features of malwares. Then feature engineering technique is used for decreasing the selected features. By embracing the notion of machine learning and datamining, we created a malware detection technique.

1.2 Motivation and Problem Statement

Buffer overflow is among the 25 Top most Dangerous Software Errors. It ranks high in the Common Weakness Enumeration. It can cause direct memory manipulations. A cautiously crafted input by a malicious actor can overwrite a register that stores important information and thus by doing so gain access to the program or in worst cases gain access to the root or admin account.

Hence, there is a need for an effective detection technique that can effectively detect heap based overflow vulnerability. Assessment of Heap based Vulnerability along with the

implementation of proposed hybrid technique can be used by the programmers to develop secure applications.

1.3 Objectives

The main objectives of thesis are: -

- Proposition of new technique for the detection of heap-based overflow.
- To create a dataset that highlights heap based overflow vulnerability in different classes of malware
- Implementation of detection technique which is the combination of both machine learning and datamining approach.

1.4 Relevance to National and Army Needs

a. **National Needs.** No industry can progress without such system that alerts them about their weaknesses. If such weaknesses are exploited by users having malicious intent it can badly affect the reputation of the industry. An efficient detection system that can detect memory attacks and can help industries to protect them self from malware that is capable of using their machines by manipulating or by corrupting the sensitive data.

b. **Military Needs.** A military data center contains confidential and important data. If the securities of such systems are compromised it will directly affect the security of our nation. A detection system that can effectively detect heap-based overflow attack can improve the security of such systems.

1.5 Thesis Contribution

It is stated that our work is unique because after extensive research we have found that there is no single dataset that gives the detailed information about the malware showing their identified class and the heap based overflow vulnerability. We have not only created a dataset but also proposed a hybrid malware detection technique.

Our contributions in this research are listed below:

- Proposition of a hybrid detection technique that is the combination of both data mining and machine learning approach
- Creation of a program to automatically extract feature from malwares files
- Creation of a program for data cleaning
- Identification of heap based overflow vulnerability of seven classes of malware which are Adware, Backdoor, Downloader, Dropper, Keylogger, Rootkit and worm.
- Three Classifiers in WEKA are trained by extracted features dataset for detection of heap based overflow vulnerability.
- Results are generated and accuracy is calculated of each classifier for our created dataset.

1.6 Thesis Organization

The thesis is structured as follows:

- Chapter 2 comprises of literature review. In this chapter evaluation of existing tools and diverse techniques proposed by different researcher have been presented that includes evidence combination techniques, smart fuzzing, concolic execution and anomaly detection.

- Chapter 3 comprises of Architecture of Proposed Malware Detection Technique. In this chapter details of proposed technique architecture is discussed. The technique is to accurately detect malwares that exploit heap based overflow vulnerability from malware binaries belonging to seven different classes by using data mining and machine learning approach.

- Chapter 4 covers the Malware Dataset Collection and Report Generation. In this chapter malware dataset creation, mechanism of online sandbox configuration and the procedure of report generation is discussed in detail.

- Chapter 5 comprises of Feature Engineering. In this chapter process of feature selection and feature extraction is discussed in detail.

- Chapter 6 covers Reverse Engineering techniques .In this chapter details of heap based overflow malware detection technique by using reverse engineering is discussed in detail.

- Chapter 7 covers Detection of Heap Based Overflow by using Extracted Features. In this chapter by using datamining techniques, extracted features are analyzed to detect heap based overflow.

- Chapter 8 covers Detection of Heap Based Overflow by using Classifiers. In this chapters three classifiers are trained and then tested to detect heap based overflow.

- Chapter 9 covers Validation and testing. It comprises of performance validation of our proposed system.

- Chapter 10 concludes the document. It comprises of conclusion and future work.

Chapter 2

Literature Review

2.1 Overview

This chapter includes the literature review of malware analysis. It discusses origin of malwares, their types followed by the evolution of malware detection techniques. These detection techniques helped in developing malware detection tools to detect malwares that exploit heap based overflow vulnerability.

2.2 Origin of Malwares

Technological advancements have made our small globe equivalent to a global village. Internet, complex computer networks along with the intelligent software advancement has become a crucial element to keep individuals, businesses and organizations together. This setup has led to an increased rate of cyber-crimes with every passing day.

Cyber criminals use malicious software to launch cyberattacks on computers to realize malicious goals. Designing malwares to meet certain goals that may include stealing of data, encrypting sensitive file, corrupting information, displaying unwanted advertisement or to gain control of a computer system. Malwares use vulnerabilities of the system to exploit data. Users are tempted into running a useful code that is often displayed on a site, malware is attached with this useful code & that it is activated on the host system.

Over time, malwares have evolved each has its unique technique to exploit the user or businesses and can be classified into categories depending upon what type of malicious activity they perform on their host.

Symantec's 2018 Internet Security Threat Report (ISTR) reveals that 600 percent increase in IoT attacks is recorded. Cryptojacking explodes by 8,500 percent, stealing resources and increasing vulnerability. It further says that malware implants grow by 200 percent, exploiting the software supply chain. Mobile malware variants are increased by 54 percent, and according to their sensors record about 126.5 million of threat events are logged every second from 157 countries and territories. Fig. 2.1 shows the pi-chart (%) of different popular malwares.

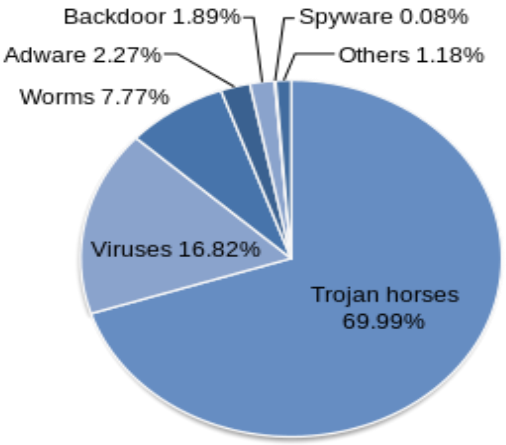


Figure 1: Popular Malwares

2.3 Malware Detection Techniques

Malware detection techniques are implemented through malware detector. The malware detector attempts to help protect the system by detecting malicious behavior. The detector may or may not reside on the same system it is trying to protect from malicious code. Using manifested malware detection techniques malware detector performs its protection, and serves as an experimental means of evaluating malware detection technique's detection capability.

Techniques used for malware detection largely categories into three parts: Static Analysis, Dynamic analysis and Hybrid analysis. The malware analysis that Anti-virus companies do, can be classified broadly into two categories; the static analysis techniques and the dynamic analysis techniques. The static techniques involve looking into the binaries directly or reverse engineering the code for patterns in the same.

The dynamic analysis techniques involve capturing the behavior of the malware sample by executing it in a sandboxed environment or by program analysis methods and then use that for extracting patterns for each family of virus.

Rossow, et al., presented a survey on literatures for malicious software detection techniques. Muazzam, et al., also presented a survey on mining techniques to detect malwares on the basis of file features. CWSandbox was proposed by Willems, et al., and it is a well-known tool which can run malware samples in a virtual environment.

Choudhary and Saharan also used data mining technique to detect malicious software. They use abstract assembly and selected top features. IDApro was used to generate the assembly code. SVM and Neural net classifiers are considered.

Malware detection tools can be categorized in to the following three groups. 1. Static Malware Detection Tools 2. Dynamic Malware Detection Tools 3. Online Malware Detection Tools. Many different types of malware detection tools are available in the market and are used with variant approaches. Their names and approaches are given as under:

i. IDA Pro Anomaly Based Approach
ii. OllyDbg Heuristic Based Approach
iii. Regshot String Matching
iv. Process Monitor Probabilistic Approach
v. Process explorer Address space randomization
vi. Virus Total Heuristic Based Approach
vii. Anubis Behavioral Based Approach
viii. Threat Expert Behavioral Based Approach
ix. Comodo Signature Based Approach

2.4 Detection of Heap-Based Overflow

Buffer overflow is a reputed software vulnerability. In the past two decades, numerous approaches have been recommended to detect this vulnerability.

Anitta Patience Namanya et al. in “Detection of Malicious Portable Executables Using Evidence Combinational Theory with Fuzzy Hashing” presented the techniques to, calculate the similarity of the Portable Executable files and according to researcher by using evidence combination techniques, detection rates can be improved [1].

Maryam Mouzarani et al. in “Smart fuzzing method for detecting stack-based buffer overflow in binary es” presented the concolic execution to determine the factors that can cause stack-based buffer overflow in binary codes [2].

Maryam Mouzarani et al. in “A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes” presented the concolic execution based smart fuzzer to determine the factors that can cause heap-based buffer overflow in executable codes [3].

Zane Markel and Michael Bilzor. in “Building a machine learning classifier for malware detection” researcher has presented a machine learning technique for the detection of malware. With the varied malware prevalence the researchers calculate variations in classifier performance [8].

S.K. Pandey et al. in “Performance of malware detection tools: A comparison” presented the evaluation of existing tools and procedures for malware detection and concluded that top three tools are Regshot, Process Monitor and Process Explorer [11].

Mikhail Zolotukhin et al. in “Detection of zero-day malware based on the analysis of opcode sequences” presented the anomaly detection technique to detect malwares. A software behavior model is proposed to detect the unseen malwares [12].

2.5 Conclusion

Many researchers has used different techniques for the detection of malwares that exploit overflow vulnerability but failed to inspect the run time performance of malwares and the methods proposed by them are also ineffective against encrypted features. In our proposed framework we have extracted and used more than twenty features for efficient

detection of malwares that exploit heap based overflow vulnerability. The propose methodology is automatic and flexible to be deployed in any operational environment.

Chapter 3

Architecture of Proposed Malware Detection Technique

3.1 Introduction

In this chapter details of proposed architecture of malware detection technique is discussed in detail. The technique is to accurately detect malwares that exploit heap based overflow vulnerability from malware binaries belonging to seven different classes by using data mining and machine learning approach.

3.2 Components of Architecture

Our proposed architecture has five main components:

a. Component 1: Feature Extraction
b. Component 2: Feature Selection
c. Component 3: Reverse Engineering
d. Component 4: Data Cleaning and Transformation
e. Component 5: Learning Algorithm

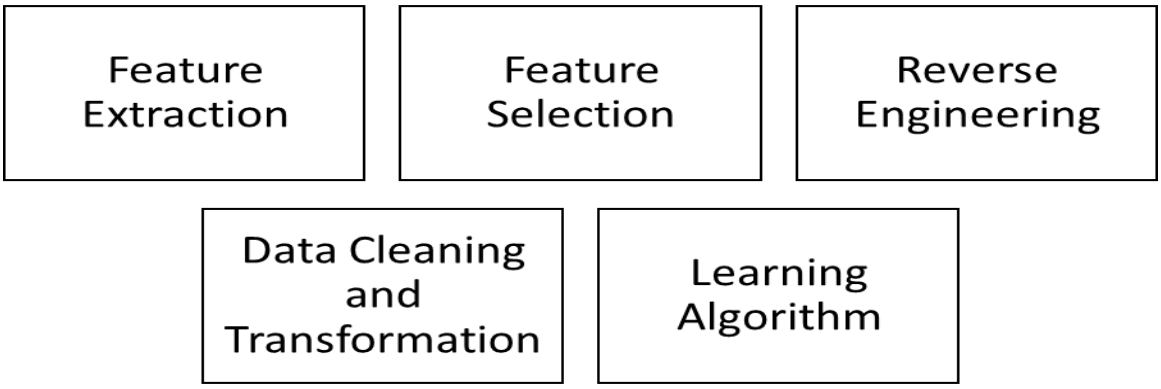


Figure 2: Components of proposed Architecture

Component 1: Feature Extraction

Feature parser extract data from all JSON files of malware dataset. JSON files are created by using IBM X-Force Exchange Tool. After extracting features, parser stores the extracted features in a CSV file.

Component 2: Feature Selection

Extracted Raw Features with calling frequencies greater then threshold are selected.

Component 3: Reverse Engineering

Type of software vulnerability has been identified by code analysis by executing malware in a sandbox environment. OllyDbg and IDA Pro tools are used for code analysis.

Component 4: Data Cleaning and Transformation

Self-Written VBA Macro code is used for data cleaning which will be explained in detail in next section.

Component 5: Learning Algorithm

Learning algorithms are used to drive a classification results from the created labeled dataset.

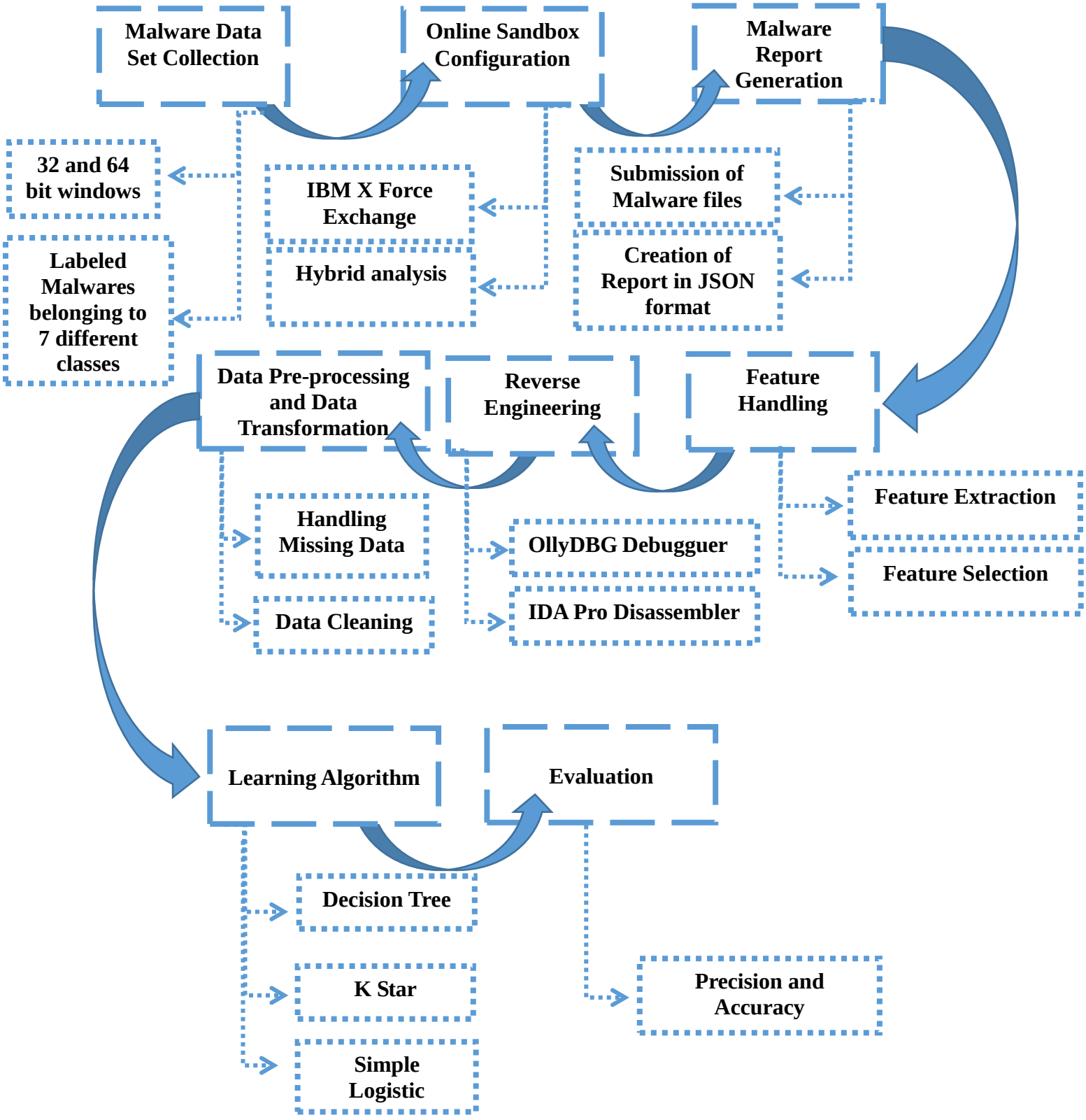


Figure 3: Architecture of Proposed Malware Detection Technique

3.3 Conclusion

This chapter describes the major components and architecture of proposed detection technique which is the combination of both datamining and machine learning approach. It gives a summarize view of complete dissertation.

Chapter 4

Malware Dataset Collection and Report Generation

4.1 Introduction

This chapter comprises of three sections that discuss in detail about the malware dataset, mechanism of online sandbox configuration and the procedure of report generation.

4.2 Malware Dataset

Malware dataset comprises of malware executables is taken from online malware and URL scanner that is from Virus Total.

Our selected Malware dataset has following characteristics:

a. Windows based Malware executables supports 32-bit and 64-bit Operating System.
b. High detection rate and can be detected by more than 20 antiviruses.
c. Labeled Malware files belonging to seven different classes including Adware, Backdoor, Downloader, Dropper, Keylogger, Rootkit and Worm.

Type of Malware	Total Count
Adware	146
Backdoor	100
Downloader	100
Dropper	96
Keylogger	110
Rootkit	96
Worm	99
Total	**747**

Table 1: Malware Dataset

4.3 Online Sandbox Configuration

To analyze malwares executables we have selected two online sandboxes that are hybrid analysis and IBM X-Force Exchange tool. Hybrid Analysis use hybrid analysis technology and falcon sandbox to analyze malware files and summarize the result by generating report. IBM X-Force Exchange tool has malware analysis components that uses cloud-based threat intelligence distribution podium to generate malware reports. Malware Analysis of IBM X-Force is a component of larger Security Operations and Response platform. Initial registration of subscription of services is required to use both the sandboxes.

4.4 Report Generation

IBM X-Force tool is used for report generation. Total 747 malware executables belonging to seven different classes have been uploaded on IBM tool for analysis. The generated report is created in JSON format. The report contains the detail analysis of each and every feature of malware and rate the malware according to its level of severity.

4.5 Conclusion

This chapter gives an overview of the selected platform and dataset for the implementation of detection technique. Platform selection is the most important step as the report generated by the selected platform will be used to extract features as discussed in detail in next Chapter. The accuracy of the results will be directly dependent on the generated reports.

Chapter 5

Feature Engineering

5.1 Introduction

One of the most important phase in machine learning is defining the suitable feature illustration. The process of feature selection and feature extraction is known as feature engineering. Feature engineering is used for converting raw data into features that present the problem to the logical models, ensuing in enhanced model accuracy on testing data. The features in dataset directly affect the learning algorithm used to predict the results. The more time and importance given to select features will result in better features. Better features results in agility, simpler model and accuracy.

5.2 Feature Extraction

Some dataset are too big in their raw state to be modeled by learning algorithms directly. So, to reduce the dimensionality of dataset, process of feature extraction is used. For feature extraction the key is that the methods used should be automatic. To automate the process of feature extraction feature parser is used. Feature parser is a self-written program in .php language.

In order to extract features from malware executables we first required the malware report in JSON format that we have generated with the help of IBM X-Force Exchange malware analysis tool. X-Force Exchange Malware Analysis is an IBM tool that can analyze multiple malware executables within a minute. The generated malware report is save in a JSON format is fed as an input to the feature parser.

5.3 Feature Parser

Twenty one features are extracted by using self-written feature parser and one feature is extracted by using reverse engineering technique. Feature parser takes multiple input files read them one by one, decode them in standard json format, extract all the required

features, save them in a tabular form and display the result of all 747 files in html file. The html file can be saved in to a CSV file both options are available for the user.

Algorithm 1: Extracting features from created JSON files

Input: JSON file created from IBM X-Force tool

Output: Extracted features from a raw JSON file

Begin

1. Read all JSON file saved in a folder.
2. Get the index of the file then decode it in a standard json format.
3. Foreach (Selected Decoded feature we get a value)

If (value == selected feature**)**

Write the feature in a tuple in a row

If (data against feature ≠empty**)**

Write all data in a next tuple of a same row

End if

End if

End for

End

5.4 Feature Selection

Feature selection is a procedure to address the problem by selecting a subset that is useful to a problem. Important features are selected to extract from malware executables. Features that are unrelated to the problem are removed. The features that are important and improves the accuracy of the model are selected. Total 21 features are selected. List of features selected from malwares executables and their total count is given in Table 2.

Extracted and Selected Feature	Total Count
Malware signature	747
Risk Parameter	79
Network Connections	179815

Number of Mutex	4924
Number of Loaded libraries	85998
Number of Process Interactions	2564
DNS Queries	1622
Process Interactions	589
Registry Writes	271
Registry Reads	274
Mutex Count	938
Mutex Name	41
File Queries	565
Type of Strings	2
Total Count of Strings	64406
Strings	284
Total Count Loaded Libraries (.dll format)	28133
Loaded Libraries (.dll format)	476
Type of Malware	7

Table 2: Extracted and Selected features

5.4.1 Malware Signature

It is a SHA-256 signature of a malware file.

5.4.2 Risk Parameter

Risk parameter is a factor of classifying and investigating potential issues that could have a negative impact on system security. There are twelve risk parameters detected in our selected dataset of malware and each risk parameter is further classified into sub categories. Bar chart representation showing the risk parameter and their total count is given below.

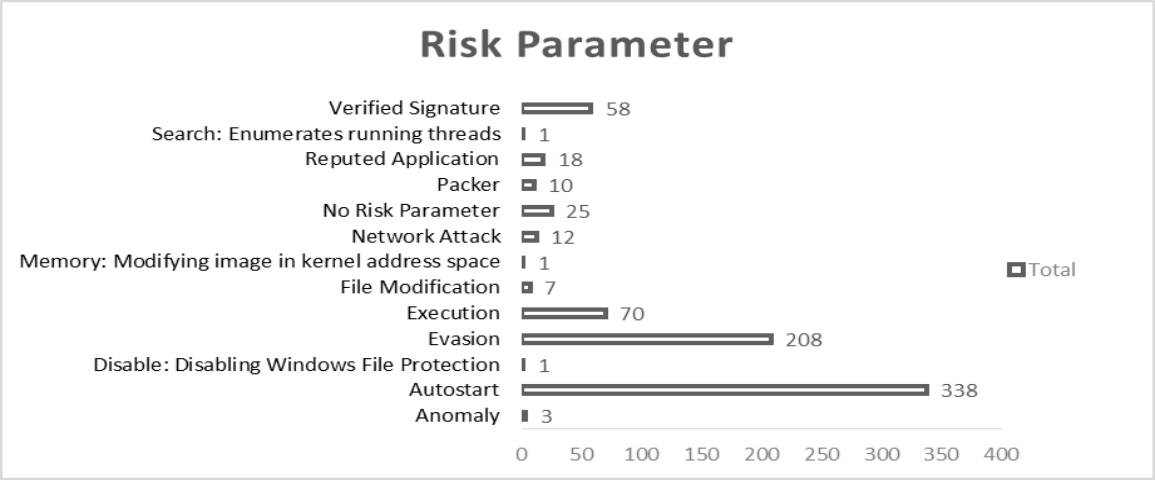


Figure 4: Risk Parameter

5.4.3 Network Connections

The Network Connection is a factor that identifies that the malware makes a network connection or not. Bar chart showing the number of network connection and their total count is given below.

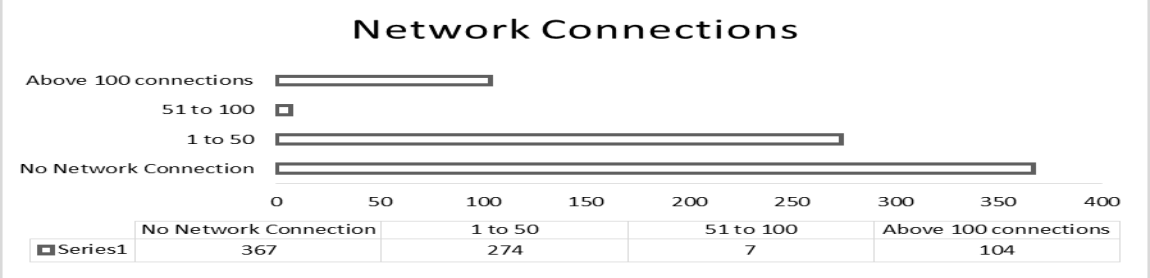


Figure 5: Range of Network connections made by different classes of malware

As per my graphical analysis of selected dataset it has been observed that downloader and worm creates many network connections which can range up to 100 and backdoor creates least number of network connections.

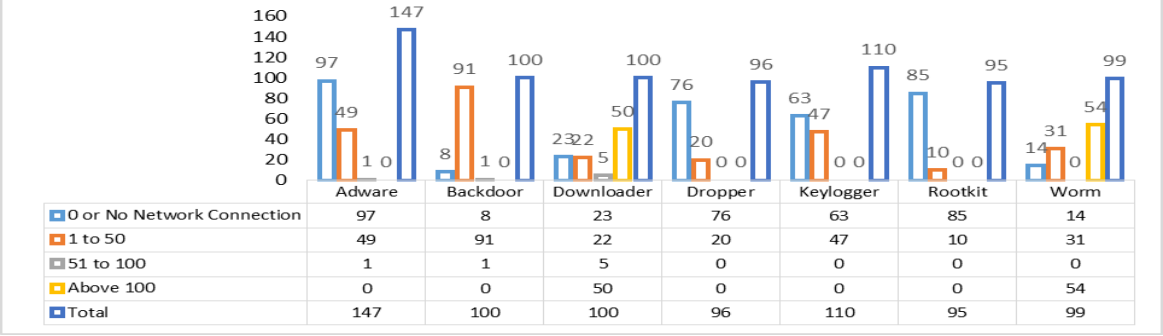


Figure 6: Network connections made by different classes of malware

5.4.4 Mutex

To synchronize access to a resource, mutex is used and is also known as locking mechanism. At a time one task that can be a thread or process based on OS abstraction can acquire the mutex. It means there is an ownership associated with mutex, and only the owner can release the lock (mutex). Graphical Analysis of extracted mutex of selected dataset is shown in a graph below.

	Adware	Backdoor	Downloader	Dropper	Keylogger	Rootkit	Worm	Count
No mutex	64	7	76	18	45	48	65	323
DBWinMutex	1	0	0	0	0	8	0	9
Global\	21	0	0	0	5	3	3	32
Local\	42	82	19	7	31	21	19	221
OSSProxyAlreadyRunning	2	0	0	0	0	0	0	2
RasPbFile	4	1	0	1	3	0	0	9
ShimCacheMutex	13	10	5	70	26	15	12	151
Total	147	100	100	96	110	95	99	747

Figure 7: Mutex used by different classes of malware

5.4.5 Process Interactions

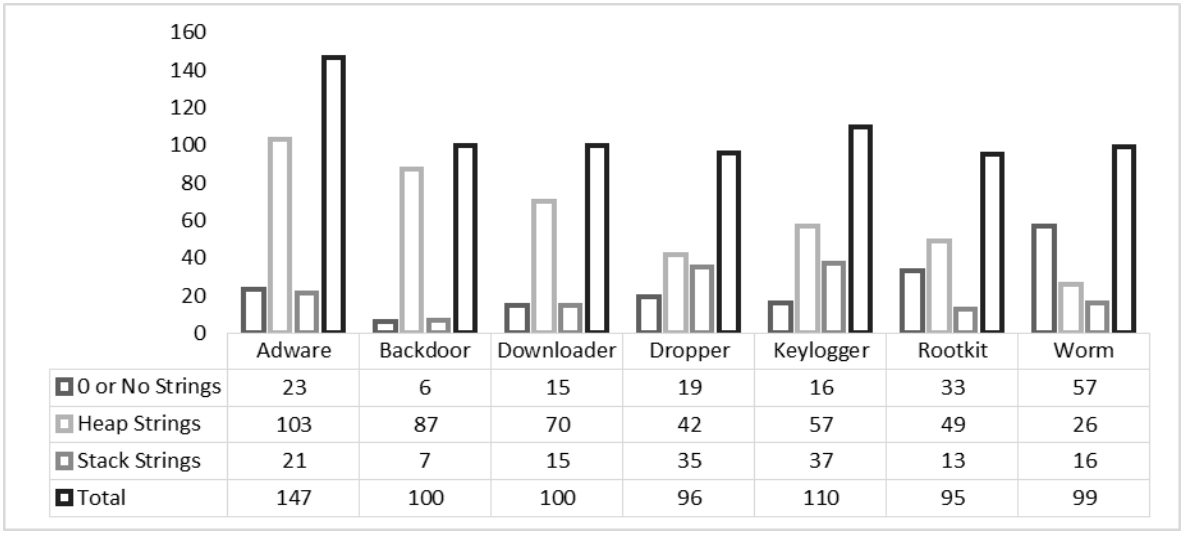
Operating systems provides a communication technique to allow different processes to communicate with each other to enable better performance and to achieve certain tasks by using code and data injection methods. Malware use process interactions to perform malicious activities. As per my graphical analysis of selected dataset it has been observed that mostly malwares perform 1 to 25 process interactions. Only downloader makes more than 100 process interactions.

	Adware	Backdoor	Downloader	Dropper	Keylogger	Rootkit	Worm	Count
0 or No process Interactions	66	0	3	4	27	47	3	150
1 to 25	81	97	93	91	81	48	94	585
26 to 50	0	2	3	1	2	0	1	9
51 to 75	0	1	0	0	0	0	1	2
75 to 100	0	0	0	0	0	0	0	0
100 to 114	0	0	1	0	0	0	0	1
Total	147	100	100	96	110	95	99	747

Figure 8: Number of Process Interactions of different classes of malware

5.4.6 Strings

String generally means an ordered arrangement of characters. It can be divided in to two sub categories i.e. Heap Strings and Stack Strings. Heap is a large pool of memory also known as dynamic memory and used for run time operation. Strings stored in a heap area are known as heap strings. As per my graphical analysis for selected data set that 24% of adware, 20% of backdoor, 16% of downloaders, 10% of droppers, 13% of keyloggers, 11% of rootkits and only 6% of worms use heap strings. So, maximum number of heap strings are used by adware and minimum number of heap strings are used by worms.



	Adware	Backdoor	Downloader	Dropper	Keylogger	Rootkit	Worm
0 or No Strings	23	6	15	19	16	33	57
Heap Strings	103	87	70	42	57	49	26
Stack Strings	21	7	15	35	37	13	16
Total	147	100	100	96	110	95	99

Figure 9: String types and their count

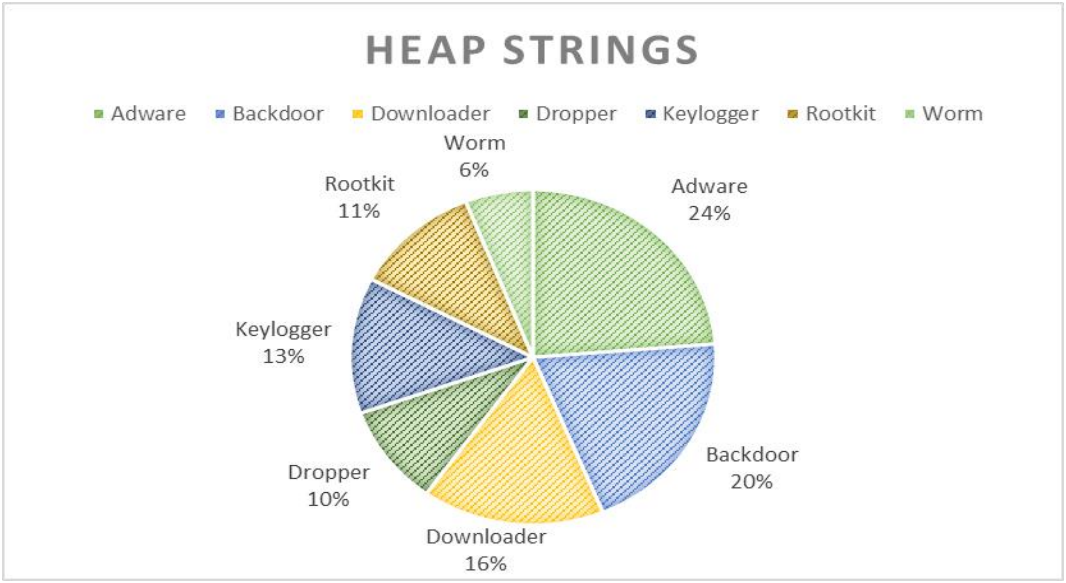


Figure 10: Heap strings belonging to different classes of malware

5.5 Conclusion

Feature parser is used to extract features from 747 malware files. Crucial features extracted with the help of feature parser are File name, Risk Parameter, Network Connections, Number of Mutex, Number of Loaded libraries, Number of Process Interactions, DNS Queries, Frequent API Calls, Downloaded Files, Process Interactions, Registry Writes, Registry Reads, Mutex Count, Mutex Name, File Queries, Type of Strings, Total Count of Strings, Strings, Total Count Loaded Libraries (.dll format), Loaded Libraries (.dll format) and Type of Malware. Feature extracted with the help of reverse engineering is overflow which will be discuss in detail in next chapters.

Chapter 6

Reverse Engineering

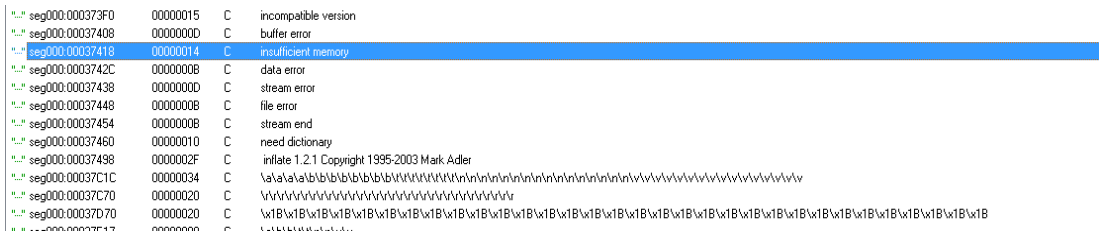
6.1 Introduction

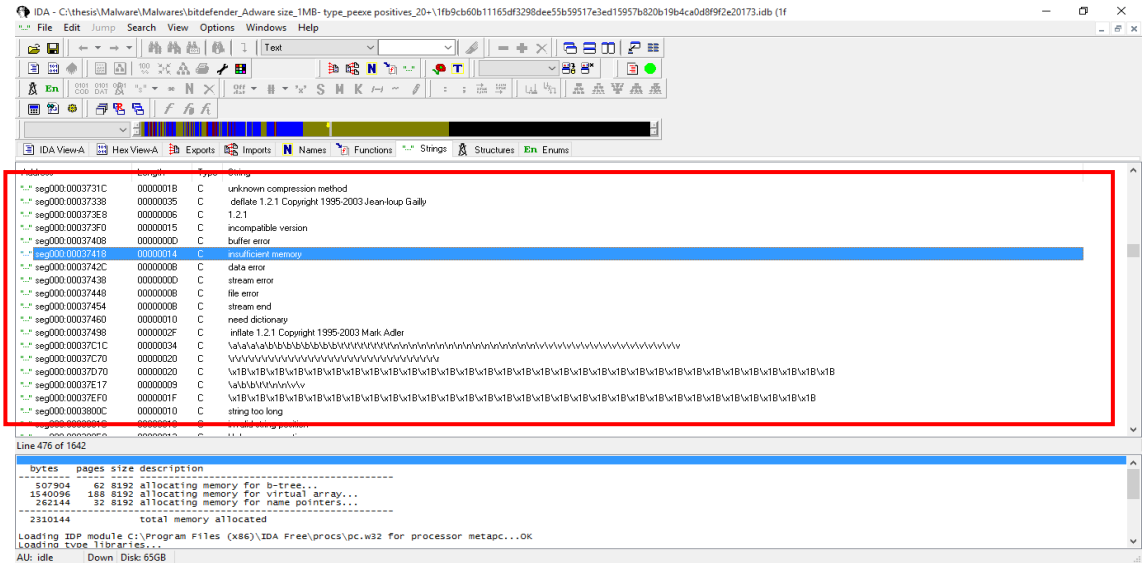
In this chapter details of heap based overflow malware detection technique is discussed in detail. The technique is to use reverse engineering tools like ollyDbg and IDA Pro to accurately detect heap based overflow vulnerability from malware binaries. Malware reverse engineering is a process to minutely explore the working of malware and to determine its effect on the environment after its execution. In our technique we have used both static and dynamic analysis for the detection of heap based overflow vulnerability in malwares.

6.2 Disassembler

IDA Pro also known as Interactive Disassembler is a disassembler which converts machine language code to assembly language source code. It is used to comprehend the functionality of the code by swapping between hex code and graph view. Its code view and string section gives a swift illustration of the mapping of flow of implementation.

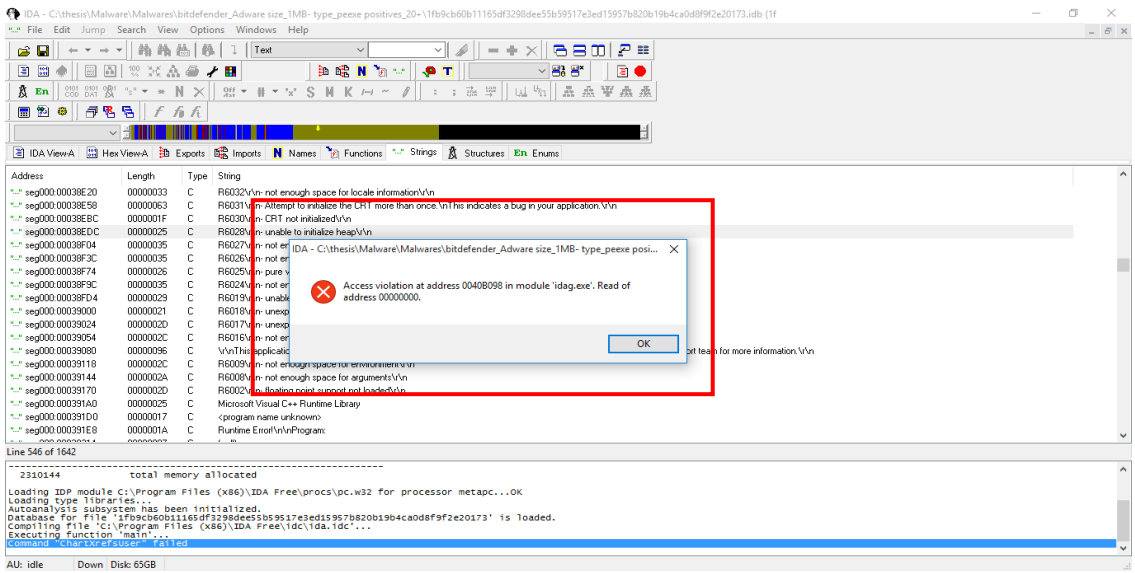
To verify that the selected malware binary file is vulnerable to heap based overflow vulnerability or not it is disassembled in to assembly source code by using IDA-Pro. To identify it disassemble an adware into assembly source code. The shah 256 hash of the selected adware is 1fb9cb60b11165df3298dee55b59517e3ed15957b820b19b4ca0d8f9f2e20173. Move to string window of the dissembled adware binary file. The string windows contain all the strings of the loaded disassembled program.





It shows that at segment 00038C44 to segment 00039080 contains run time error strings. At segment 00037418 it shows insufficient memory string.

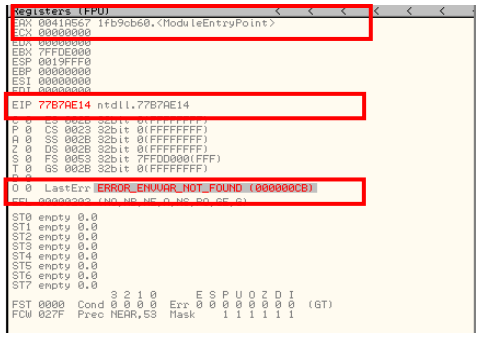
When we try to execute the assembly code in IDA Pro it gives an access violation. The access violation exception occurs only when an unapproved application attempts to access unauthorized memory fragments of your system.

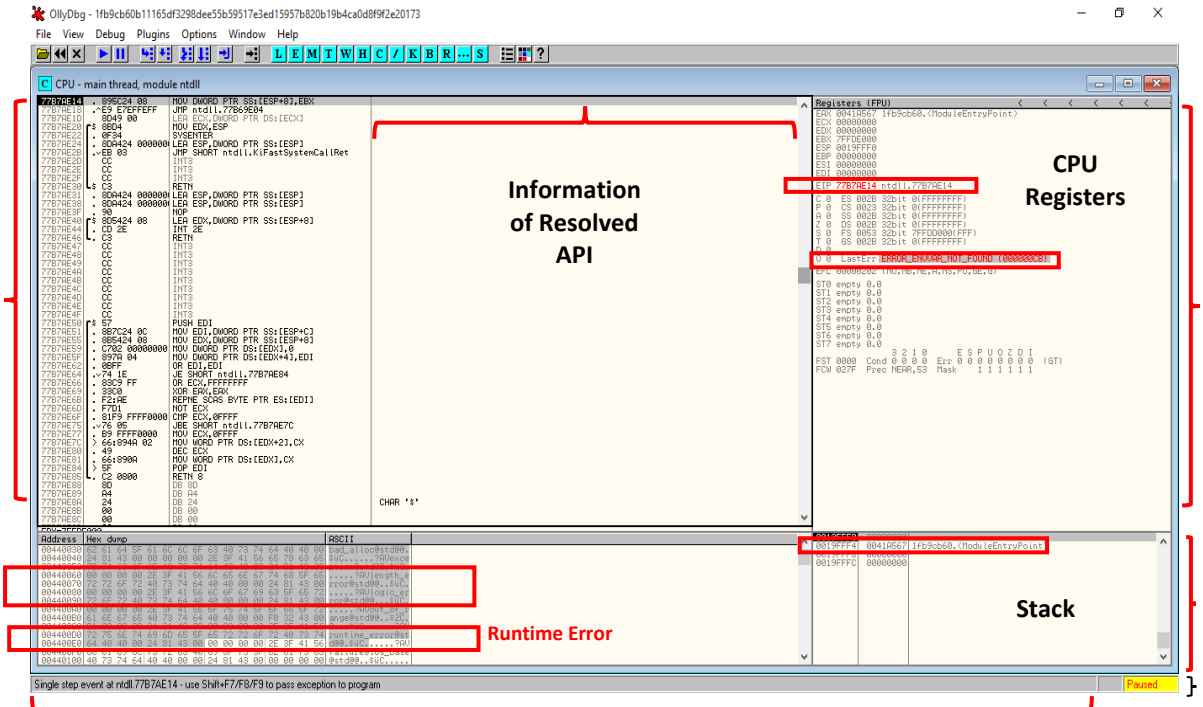


6.3 Debugger

Olly Debugger is an assembler-level investigating Debugger. It is used to analyze binary codes and to execute the application in a controlled environment that is useful to find and list the effects of malicious binaries on an environment.

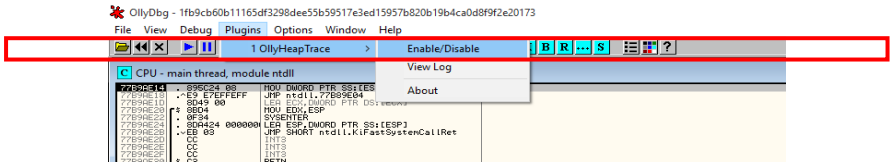
To verify that the selected malware binary file is vulnerable to heap based overflow vulnerability or not load the malware binary file in to primary memory by using olly debugger. When it is loaded the value of EIP=770EAE14 and the value of EAX register is at stack position 0019FFF4. The shah 256 hash of the selected adware is 1fb9cb60b11165df3298dee55b59517e3ed15957b820b19b4ca0d8f9f2e20173.



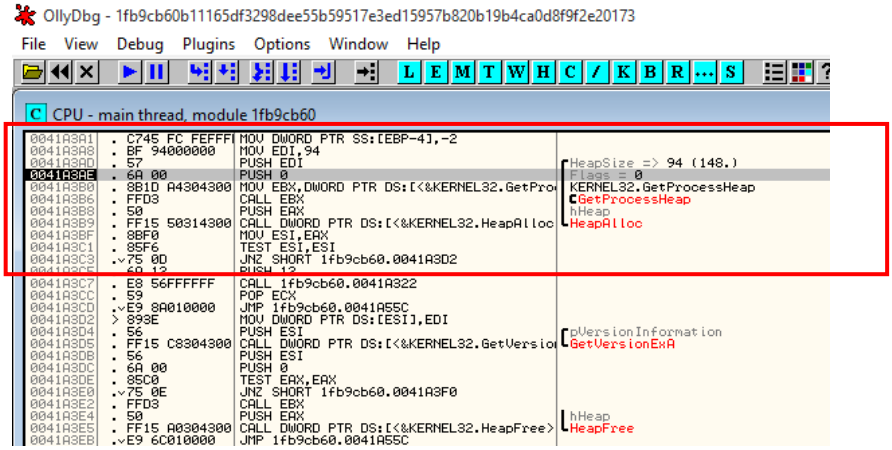


The flag register hold the information of official flags of processing in the central processing unit. These flags are affected by the processes in the Arithmetic logic unit. They are changed as a result of arithmetic and logic process. Flag O is an Overflow flag and it holds a flag resulting from an instruction that is needed in order to make decision in another instruction. Initially when an adware binary is loaded it shows LastErr ERROR_ENVVAR_NOT_FOUND (000000CB). This error occurs when the system does not find the environment option that was entered.

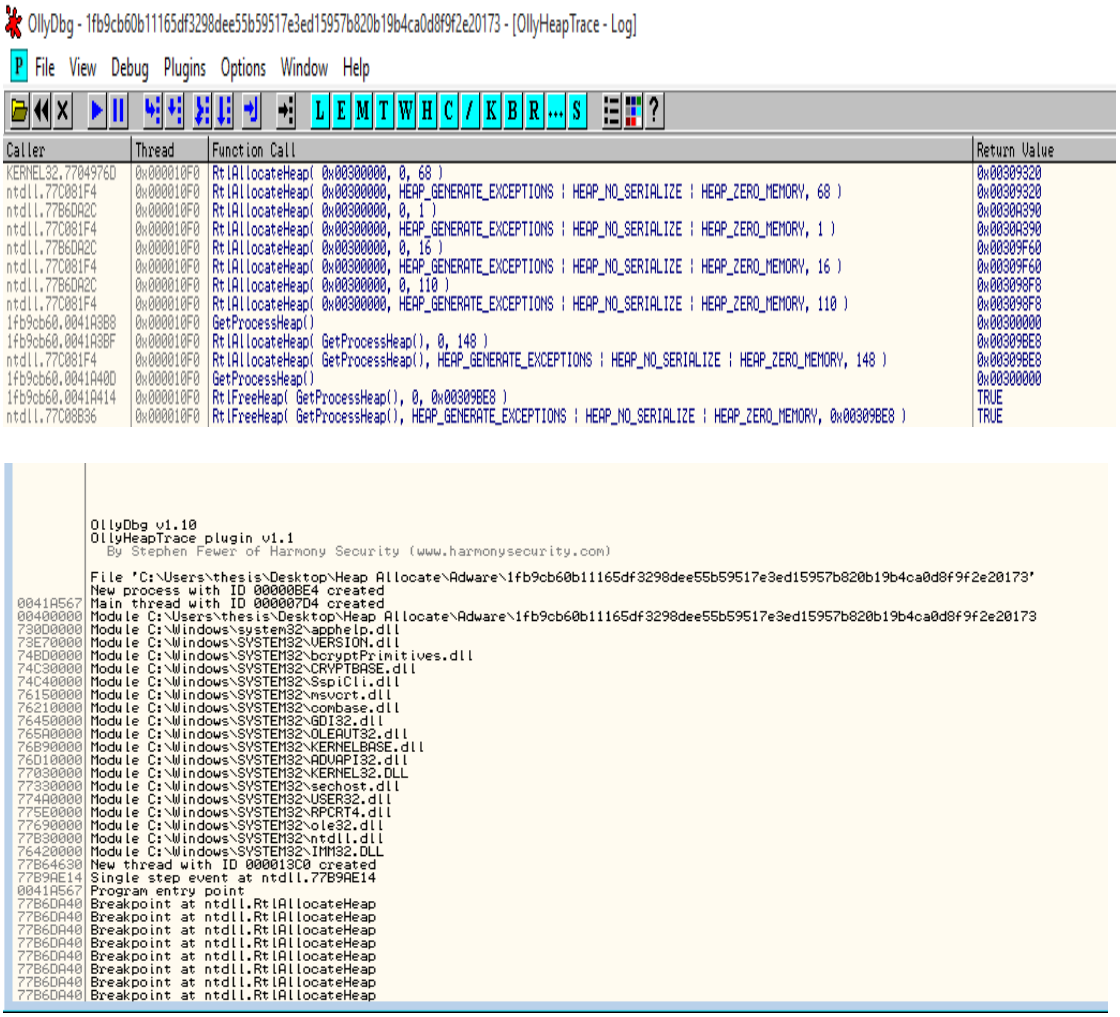
Enable OllyHeap trace and execute the code. OllyHeap trace is a plugin by Stephen Fewer. It is used to view heap allocations and deallocations for multiple heaps, as well as operations such as creating and destroying heaps. OllyHeap trace is used to depict the behavior of heap trace. Initially no heap is allocated till address 0041A39B.



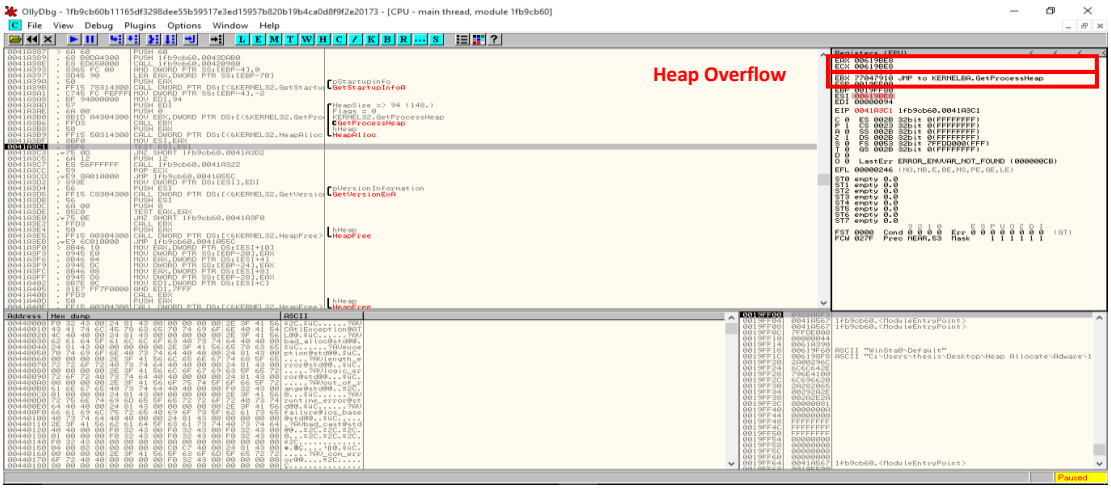
First heap allocation takes place when EIP value is equal to 0041A3A1. The allocated size of heap chunk is 94. The GetProcessHeap function is used to get a handle to the default heap for the calling process. The handle can be used by the process to assign memory from the process heap without having to first create a private heap.



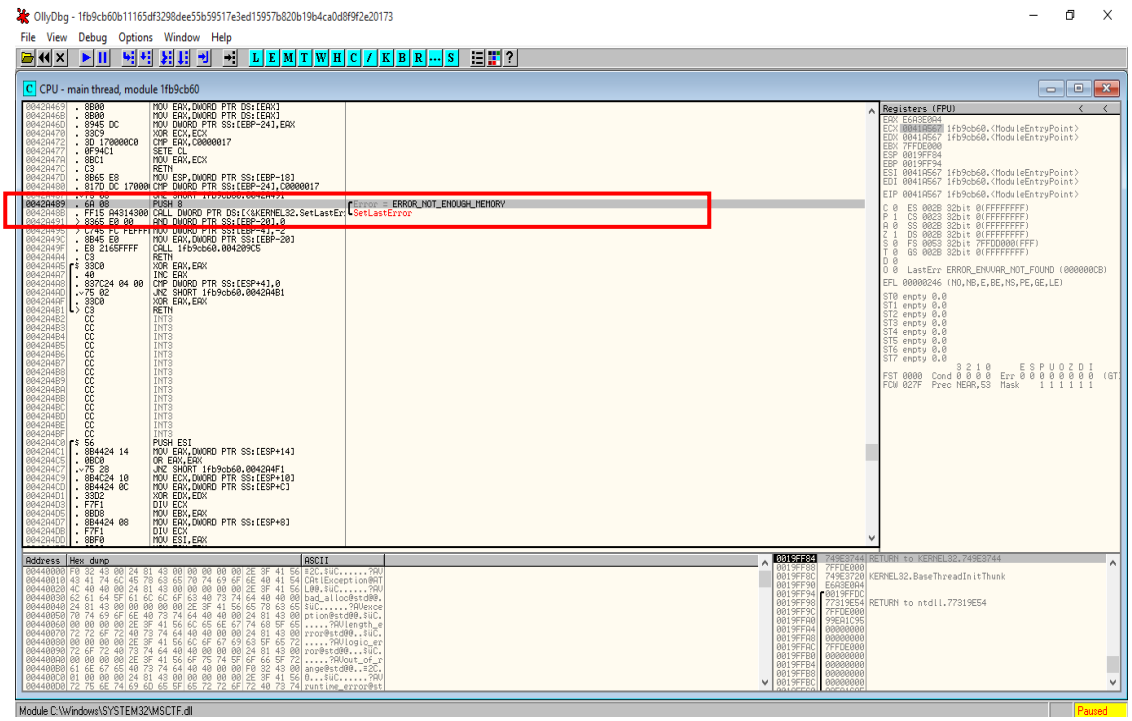
The 32-bit Kernel32 dynamic link library found in the kernel of Windows operating system. It is used to manage memory, I/O operations, and interrupts. kernel32.dll is located in protected memory space. The code at address 004A3A8 shows that get heap buffer of size 94 that is 148 bytes from protected memory. If a string larger than 147 bytes passed to it will thereby overwrite the data coincidental to this memory block which is, actually, a header of the following memory block because memory is allocated without bound checking. At this point following heap traces are generated.



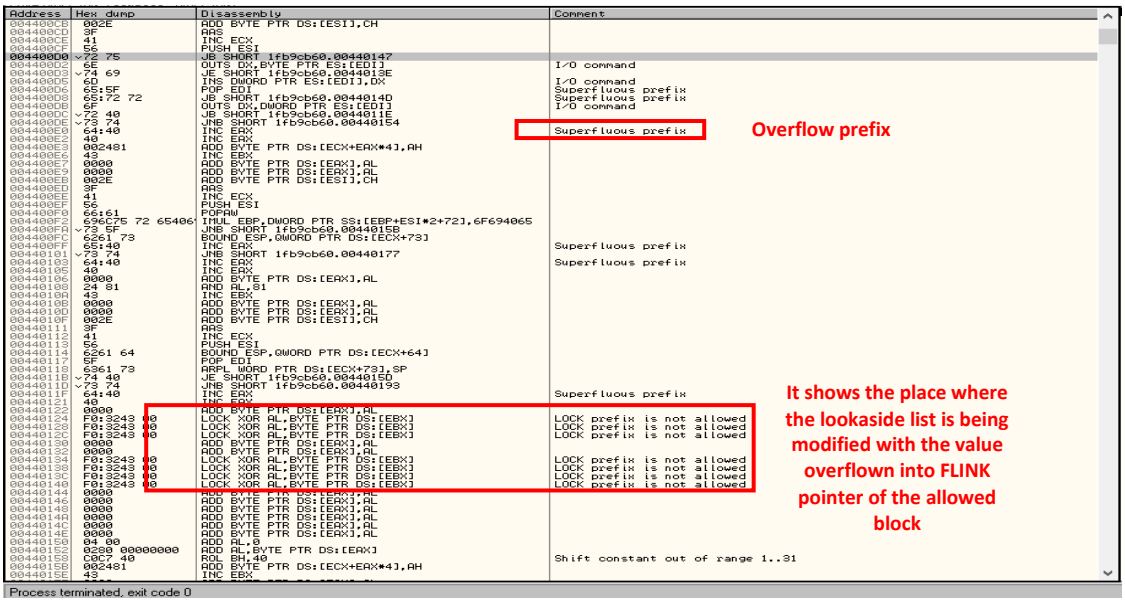
At EIP 0041A3BF Heap overflow takes place. The value which is overflown to EBX is 77047910 which points to value at stack. EBX is a non-volatile register which has no particular use but mostly used to set a common value to speed up calculations.



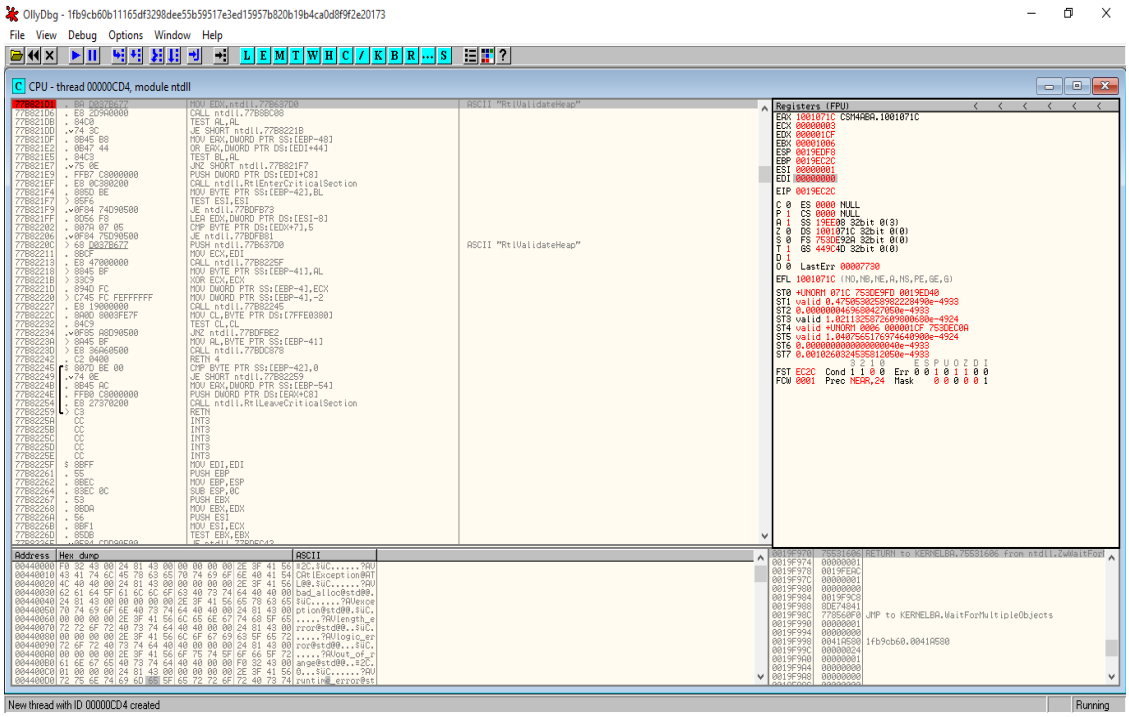
At virtual address 0042A489 we can see the error in resolved API information window. Error_NOT_ENOUGH_MEMORY this error occurs when the free space is not available or if the memory is fragmented.



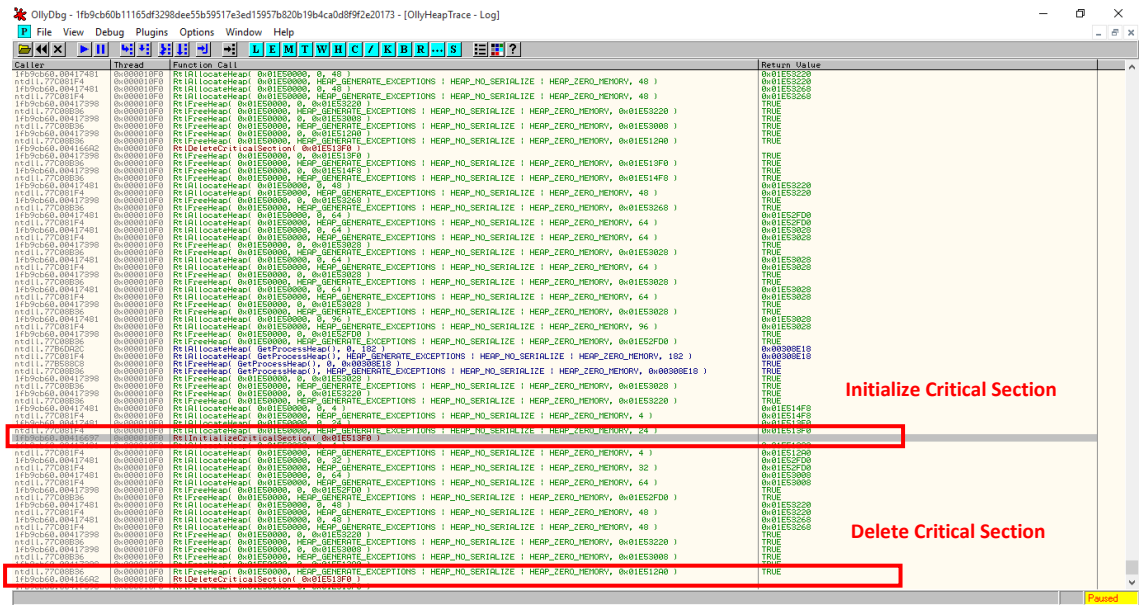
The lookaside list contains the information of heap buffers it has two pointers pointing before and after the lookaside entries. These pointers are FLINK and BLINK.



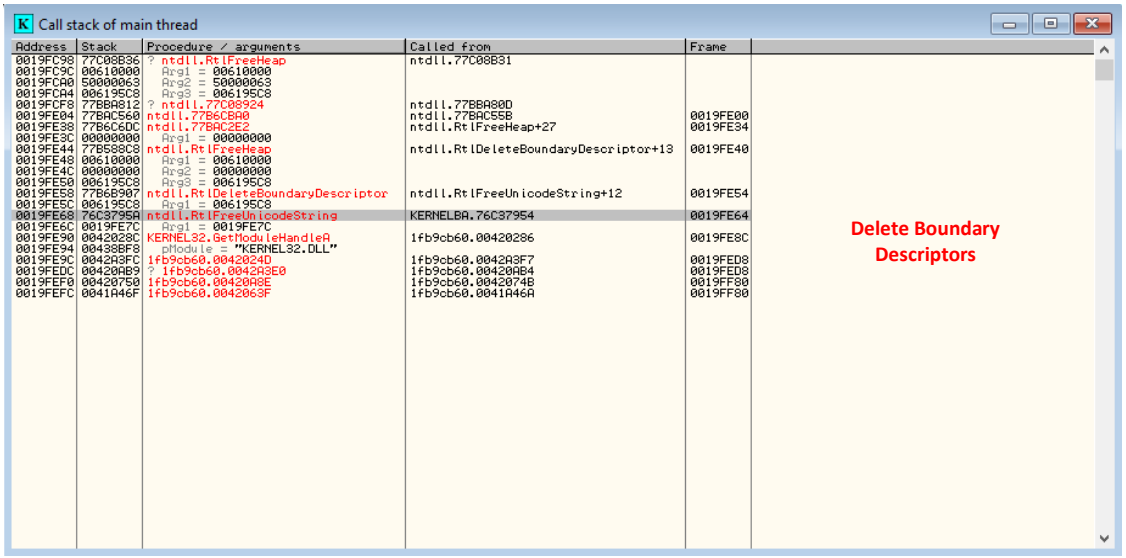
After complete execution of adware when we observe CPU window it appears as follow:



The complete Heap trace is given below:



The call stack window appears after complete execution as given below:



6.4 Conclusion

In reverse engineering the most observable method is to entirely engineer a malware but this obviously takes a great amount of time. So, in an environment where there are more than 100's of malware this approach is not practical. Another approach to detect malwares that exploit overflow vulnerabilities in big dataset efficiently with respect to time is, to use datamining technique for malware detection.

Chapter 7

Detection of Heap Based Overflow by using Extracted Features

7.1 Introduction

By adopting the concepts of datamining techniques extracted features are analyzed to detect whether the file under observation is vulnerable to heap based overflow vulnerability or not. Detailed Graphical analysis of each extracted features of 747 malware file of selected dataset is inspected in detail and rules are created for the efficient detection of heap based overflow.

7.2 Risk Parameter

Risk Parameter is a factor of categorizing and examining possible issues that could have an undesirable impact on system security. Malwares that exploit heap based overflow vulnerability use six risk parameters and each risk parameter is further classified into sub categories. Pie chart representation showing that about 63% of such malwares use autostart, 7% use execution and evasion, 5% has signatures, 1% generates traffic and use popular applications to exploit heap vulnerability.

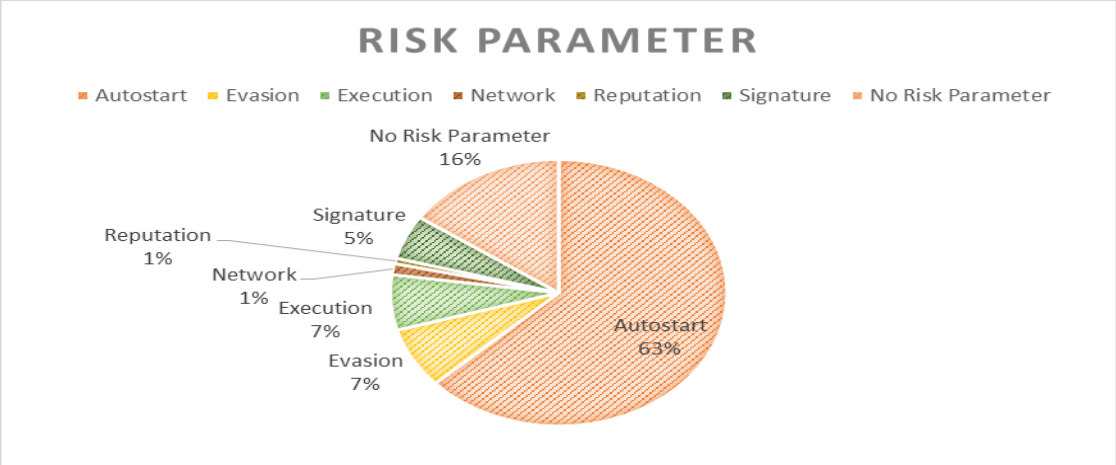
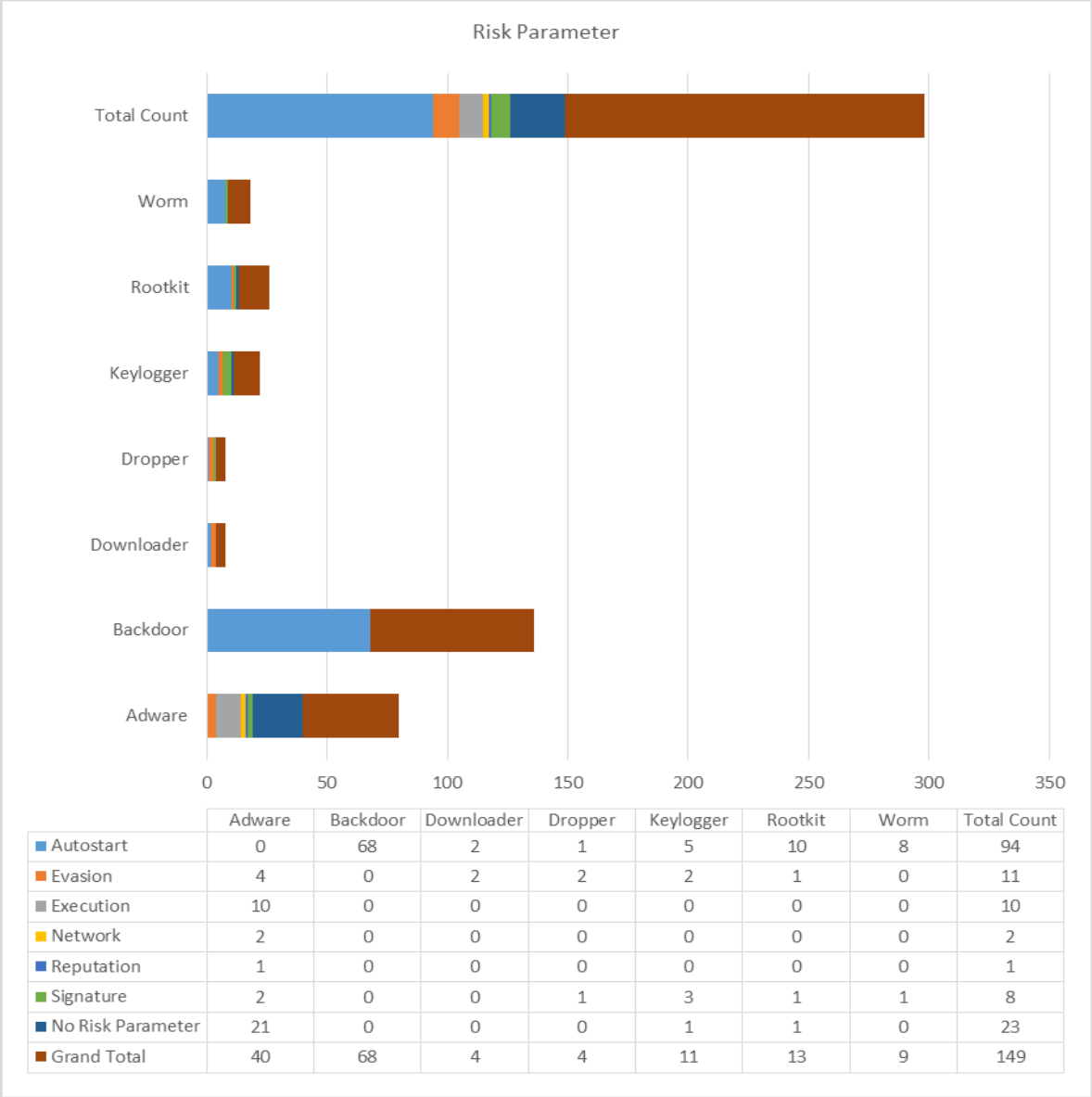


Figure 11: Risk Parameter of malwares that exploit heap based overflow vulnerability



	Adware	Backdoor	Downloader	Dropper	Keylogger	Rootkit	Worm	Total Count
Autostart	0	68	2	1	5	10	8	94
Evasion	4	0	2	2	2	1	0	11
Execution	10	0	0	0	0	0	0	10
Network	2	0	0	0	0	0	0	2
Reputation	1	0	0	0	0	0	0	1
Signature	2	0	0	1	3	1	1	8
No Risk Parameter	21	0	0	0	1	1	0	23
Grand Total	40	68	4	4	11	13	9	149

Figure 12: Count of Risk Parameter of malwares that exploit heap based overflow vulnerability

Subcategories of the risk parameter and their total count is shown in the bar chart representation given below. There are total nineteen subcategories that belongs to six different categories of risk parameter. About 60% of malwares that exploit heap based overflow vulnerability use autostart risk parameters subcategory that is registering for autostart during Windows boot. This features set the malware to run on start up. This means the malware does not need the permission to execute it and will start automatically after windows boot.

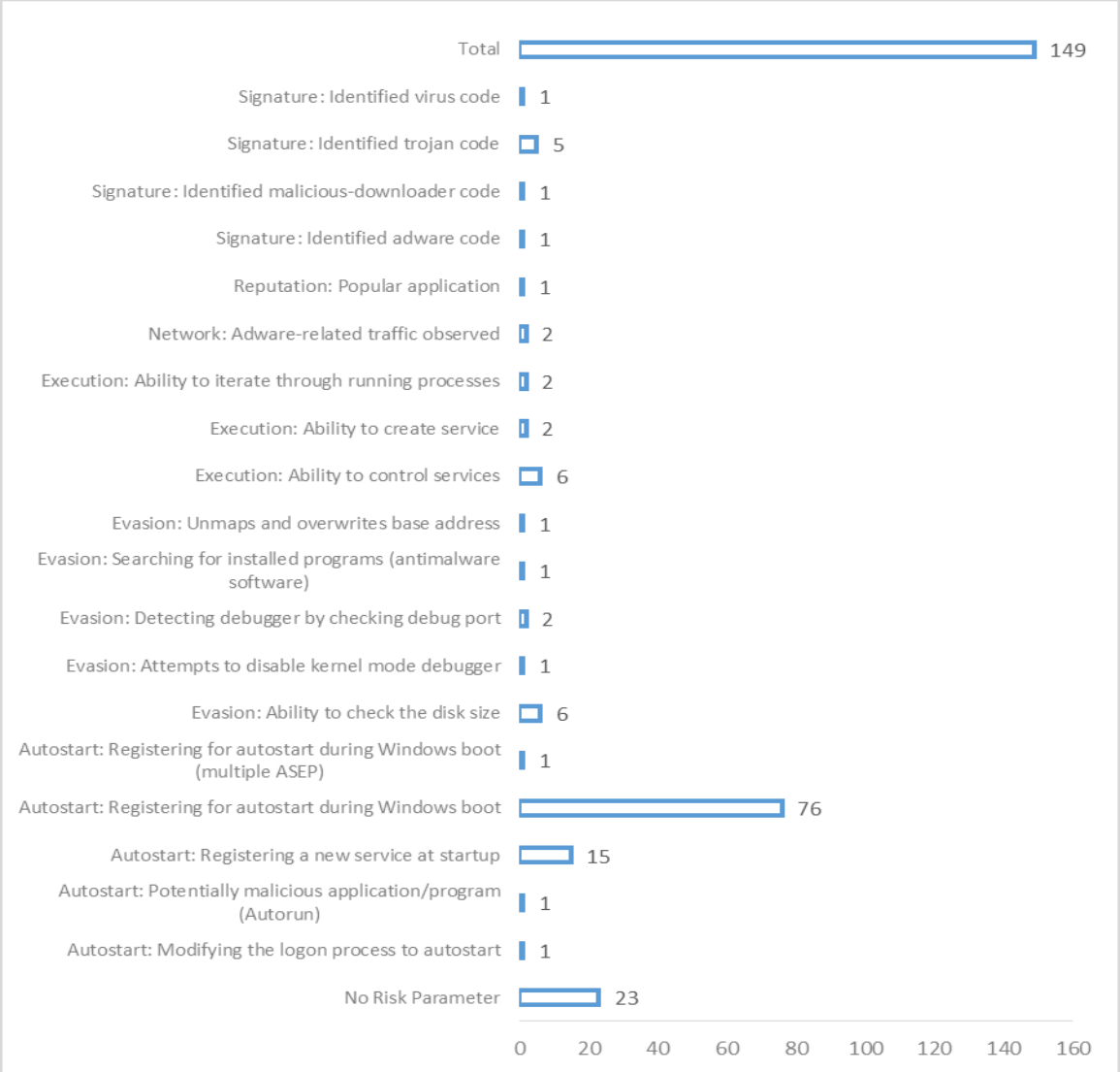
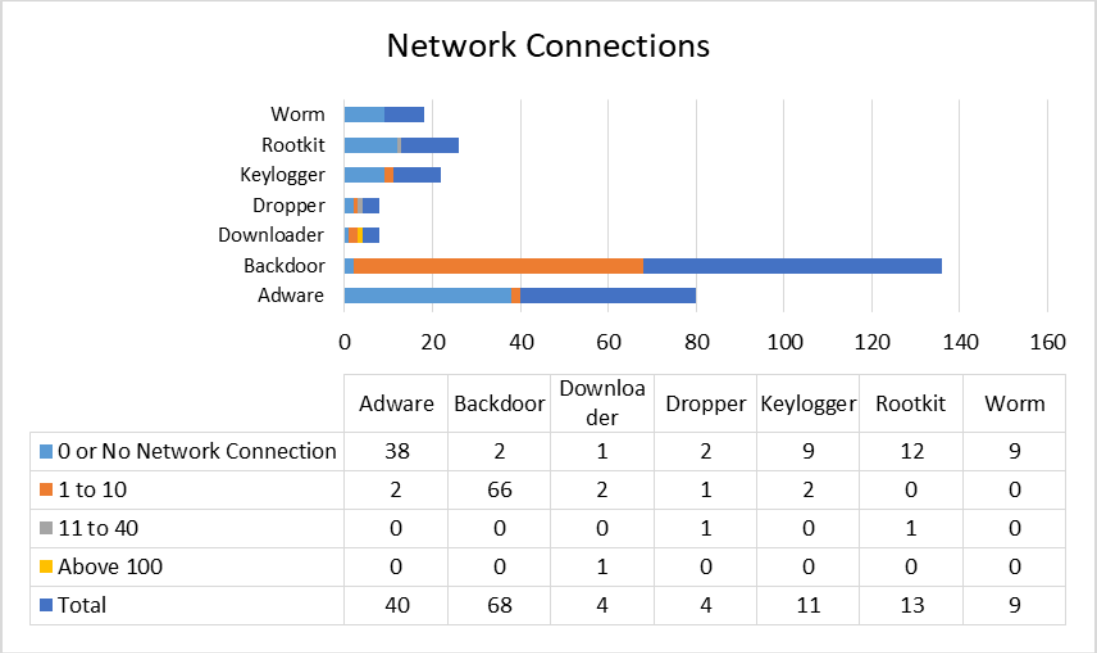


Figure 13: Subcategories of Risk parameters

7.3 Network Connection

A factor that determines that the malware makes a network connection or not. About 50% of malwares that exploit heap based overflow vulnerability makes network connection ranging between 1 to10. Bar chart representation of malwares showing network connections belonging to seven different classes is given below.

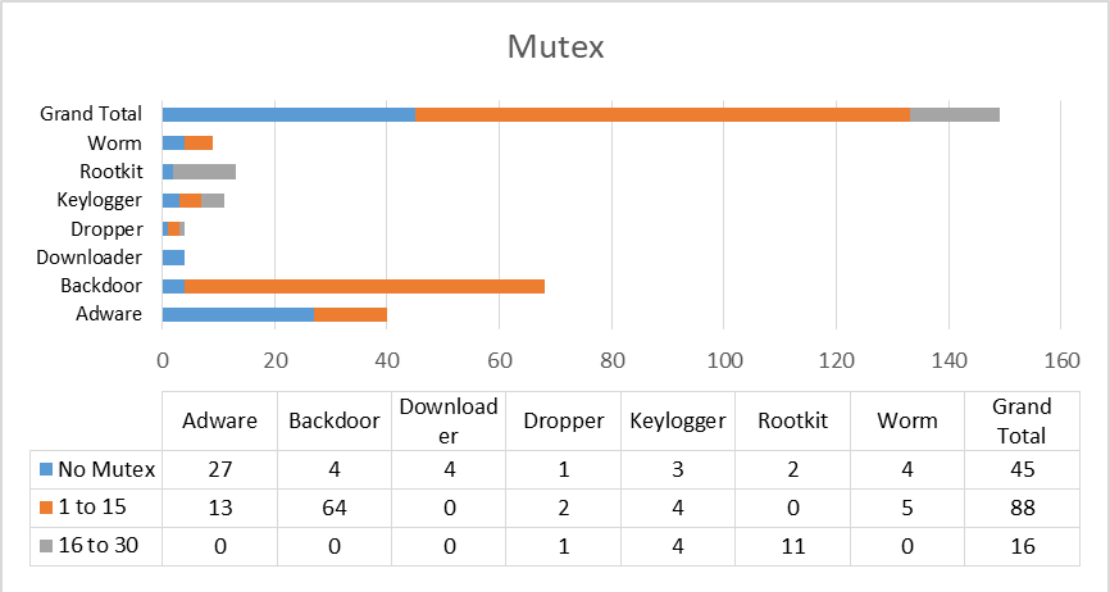


	Adware	Backdoor	Downloader	Dropper	Keylogger	Rootkit	Worm
0 or No Network Connection	38	2	1	2	9	12	9
1 to 10	2	66	2	1	2	0	0
11 to 40	0	0	0	1	0	1	0
Above 100	0	0	1	0	0	0	0
Total	40	68	4	4	11	13	9

Figure 14: Range of Network Connection of malwares that exploit heap based overflow vulnerability

7.4 Mutex

Mutex is a locking mechanism to synchronize access to resources. About 60% of malwares that exploit heap based overflow vulnerability use mutex ranging between 1 to 15. Bar chart representation of seven different classes of malwares showing mutex range is shown below.



	Adware	Backdoor	Downloader	Dropper	Keylogger	Rootkit	Worm	Grand Total
No Mutex	27	4	4	1	3	2	4	45
1 to 15	13	64	0	2	4	0	5	88
16 to 30	0	0	0	1	4	11	0	16

Figure 15: Mutex range of malwares that exploit heap based overflow vulnerability

Malwares that exploit heap based overflow vulnerability use five mutex and each mutex parameter is further classified into sub categories.

Mutex Name	A	B	D	Dr	K	R	W	Total
No Mutex	27	4	4	1	3	2	4	45
DBWinMutex	0	0	0	0	0	6	0	6
Global\	0	0	0	0	1	0	3	4
Local\	8	63	0	1	5	1	2	80
OSSProxyAlreadyRunning	2	0	0	0	0	0	0	2
ShimCacheMutex	3	1	0	2	2	4	0	12
Grand Total	**40**	**68**	**4**	**4**	**11**	**13**	**9**	**149**

Bar chart representation of mutex is given below.

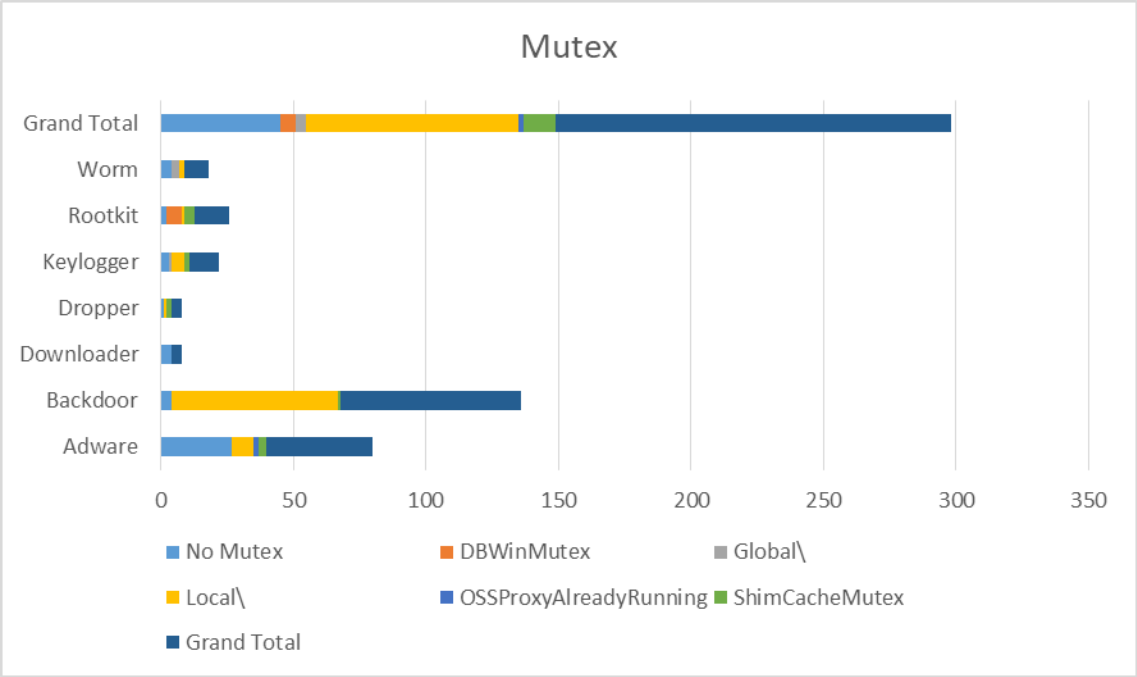
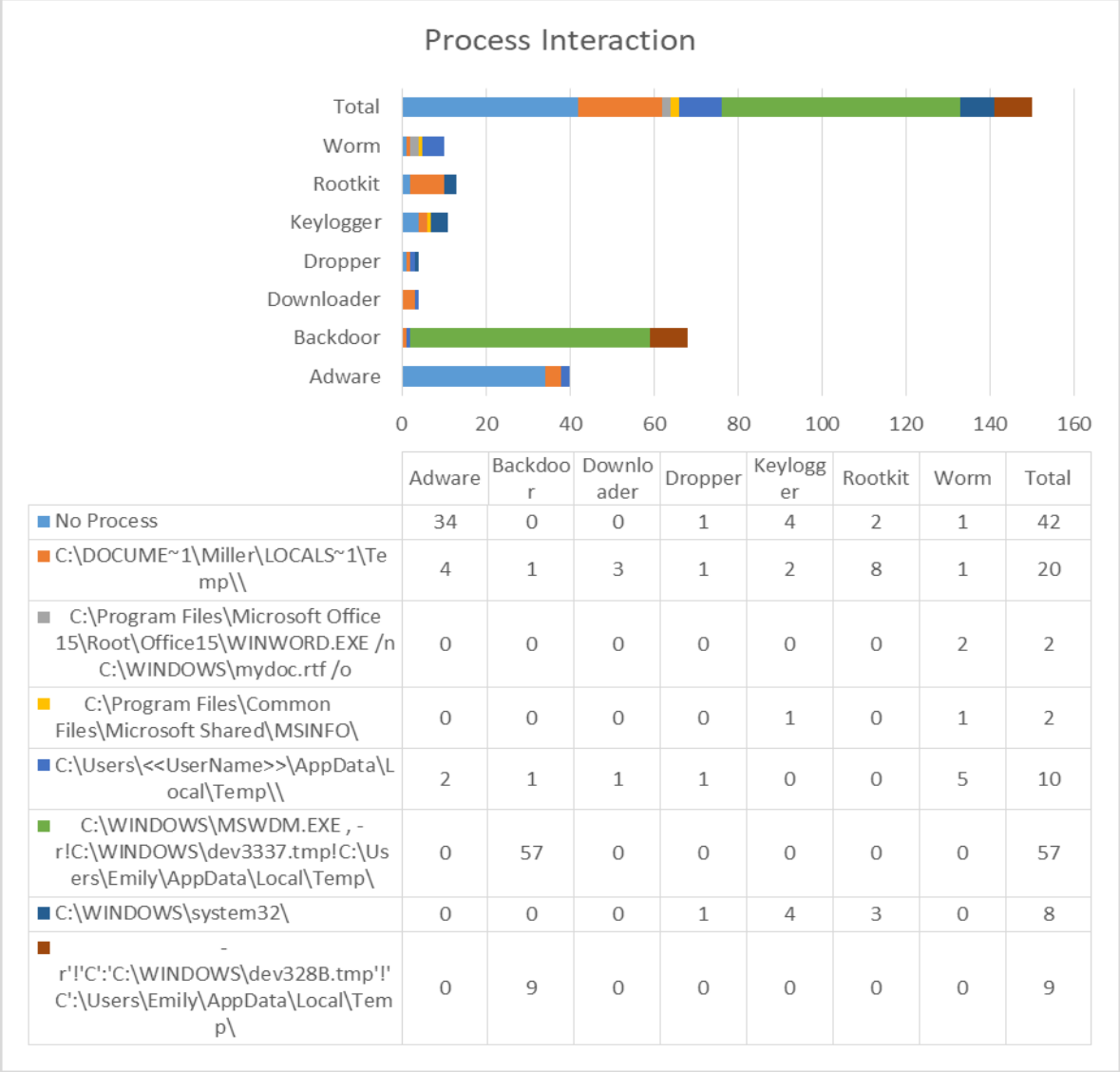


Figure 16: Mutex belonging to seven different classes of malware that exploit heap based overflow vulnerability

7.5 Process Interactions

Malware use process interactions to perform malicious activities. Most common processes used by malwares that exploits heap based overflow vulnerability are shown in bar chart representation given below.



	Adware	Backdoor	Downloader	Dropper	Keylogger	Rootkit	Worm	Total
No Process	34	0	0	1	4	2	1	42
C:\DOCUME~1\Miller\LOCALS~1\Temp\\	4	1	3	1	2	8	1	20
C:\Program Files\Microsoft Office 15\Root\Office15\WINWORD.EXE /n C:\WINDOWS\mydoc.rtf /o	0	0	0	0	0	0	2	2
C:\Program Files\Common Files\Microsoft Shared\MSINFO\	0	0	0	0	1	0	1	2
C:\Users\<<UserName>>\AppData\Local\Temp\\	2	1	1	1	0	0	5	10
C:\WINDOWS\MSWDM.EXE , -r!C:\WINDOWS\dev3337.tmp!C:\Users\Emily\AppData\Local\Temp\	0	57	0	0	0	0	0	57
C:\WINDOWS\system32\	0	0	0	1	4	3	0	8
-r'!'C':'C:\WINDOWS\dev328B.tmp'!'C':\Users\Emily\AppData\Local\Temp\	0	9	0	0	0	0	0	9

Figure 17: Process Interactions belonging to seven different classes of malware that exploit heap based overflow vulnerability

As per my graphical analysis of selected dataset it has been observed that mostly malwares that exploits heap based overflow vulnerability perform 1 to 15 process interactions. No such malwares makes more than 55 process interactions. Maximum number of process interactions are made by backdoor malwares.

	A	B	D	Dr	K	R	W
0	34	0	0	1	4	2	1
1 to 15	6	68	3	3	7	11	7
45 to 55	0	0	1	0	0	0	1

Pie chart representation showing the percentage of malwares is shown below. As per the graphical analysis it has been observed that most malwares use C:\DOCUME~1\Miller\LOCALS~1\Temp\\ path to perform malicious activities.

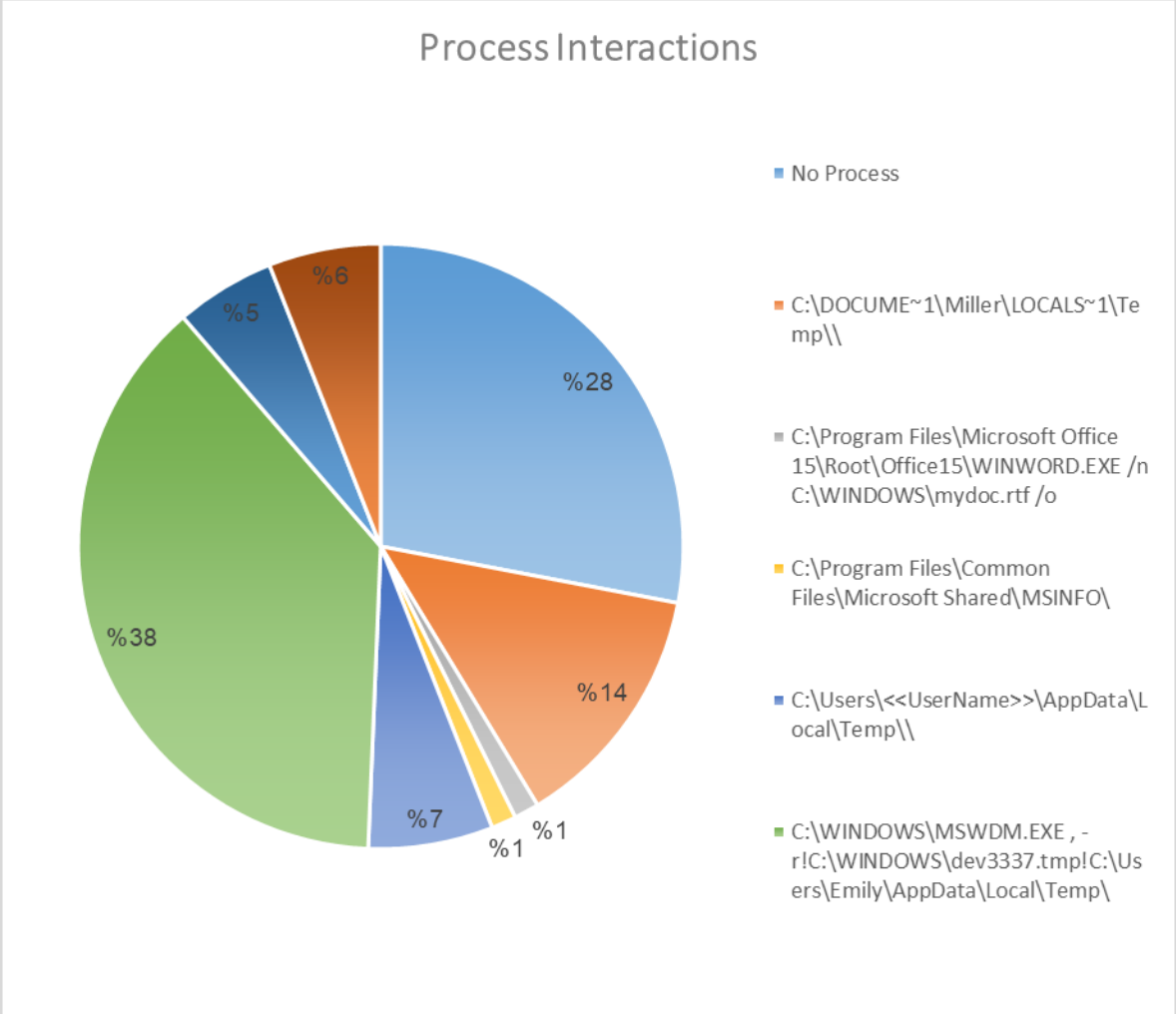
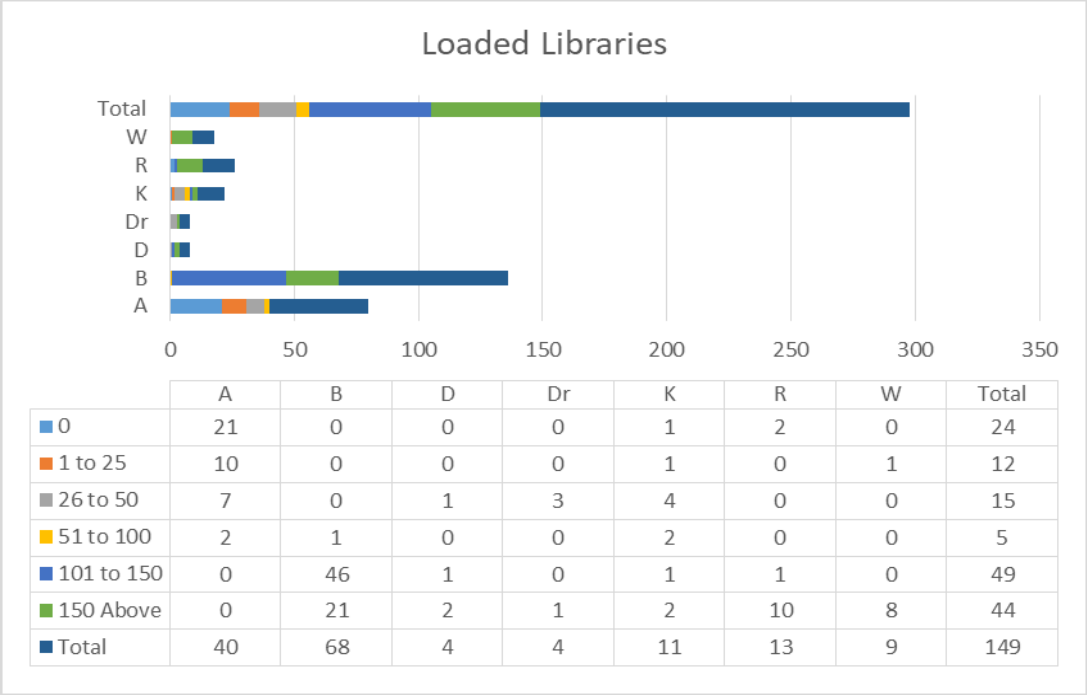


Figure 18: Name of Process Interactions belonging to seven different classes of malware that exploit heap based overflow vulnerability

7.6 Loaded Libraries

When a program running on the computer system needs a library to execute a subroutine is loaded from the dynamic load library into main memory. As per my graphical analysis of selected dataset it has been observed that mostly malwares that exploits heap based overflow vulnerability loads 101 to 150 libraries from dynamic load library.

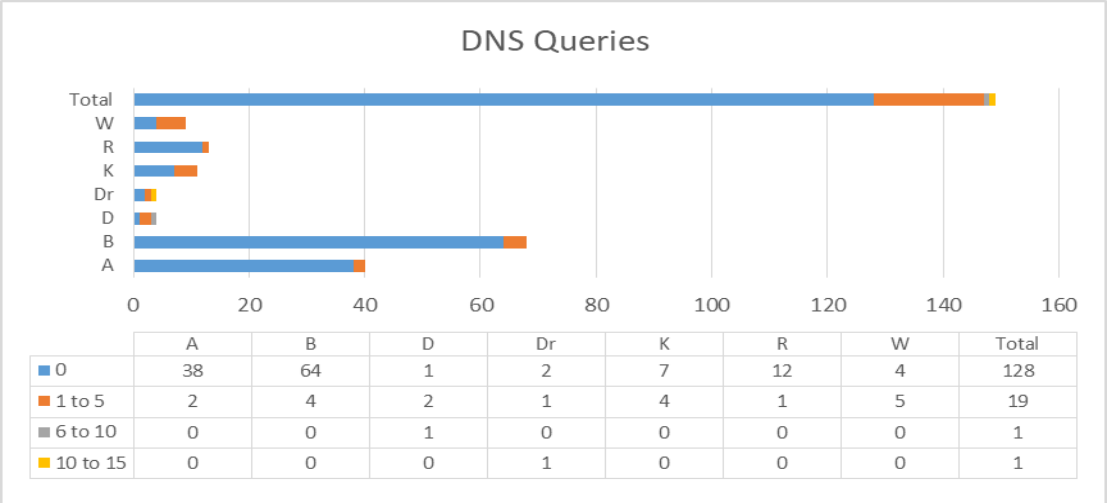


	A	B	D	Dr	K	R	W	Total
0	21	0	0	0	1	2	0	24
1 to 25	10	0	0	0	1	0	1	12
26 to 50	7	0	1	3	4	0	0	15
51 to 100	2	1	0	0	2	0	0	5
101 to 150	0	46	1	0	1	1	0	49
150 Above	0	21	2	1	2	10	8	44
Total	40	68	4	4	11	13	9	149

Figure 19: Loaded Libraries belonging to seven different classes of malware that exploit heap based overflow vulnerability

7.7 DNS Queries

To get an IP address against the DNS name an inquiry is made by the computers system or networking device. As per my graphical analysis of selected dataset it has been observed that only few malwares that exploits heap based overflow vulnerability perform DNS queries. Bar chart representation is given below.

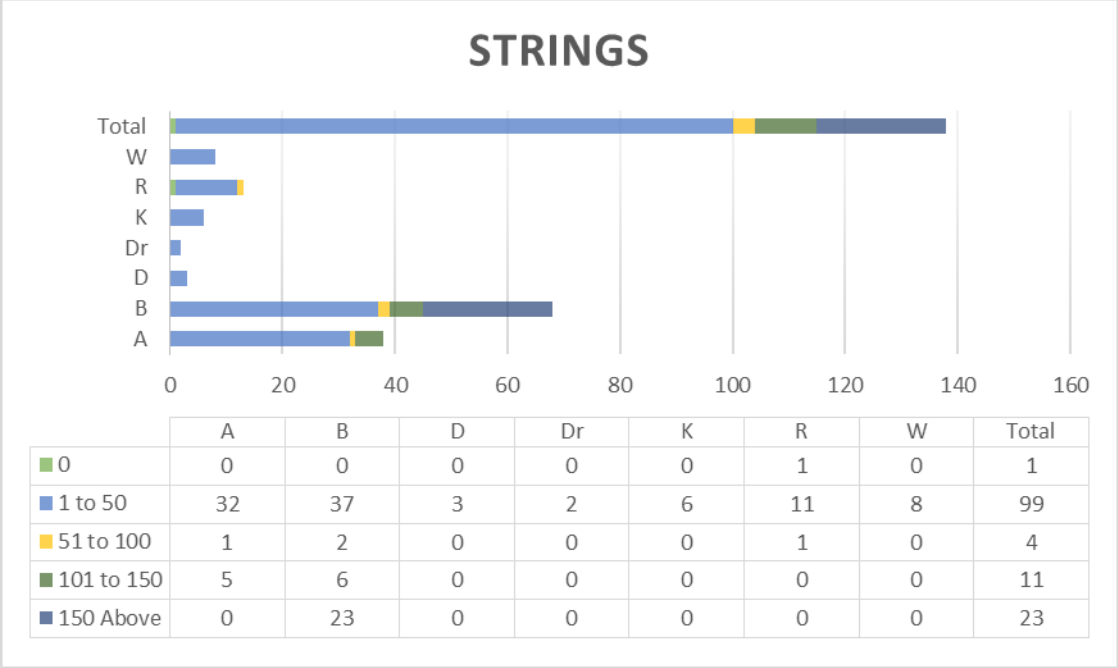


	A	B	D	Dr	K	R	W	Total
0	38	64	1	2	7	12	4	128
1 to 5	2	4	2	1	4	1	5	19
6 to 10	0	0	1	0	0	0	0	1
10 to 15	0	0	0	1	0	0	0	1

Figure 20: DNS Queries belonging to seven different classes of malware that exploit heap based overflow vulnerability

7.8 Strings

Strings stored in a heap area are known as heap strings. As per my graphical analysis for selected data set it has been observed that most of the malwares that exploits heap based overflow vulnerability has heap strings range between 1 to 50. Bar chart representation is given below.



	A	B	D	Dr	K	R	W	Total
0	0	0	0	0	0	1	0	1
1 to 50	32	37	3	2	6	11	8	99
51 to 100	1	2	0	0	0	1	0	4
101 to 150	5	6	0	0	0	0	0	11
150 Above	0	23	0	0	0	0	0	23

Figure 21: Strings belonging to seven different classes of malware that exploit heap based overflow vulnerability

7.9 Conclusion

As per above graphical analysis of features the following rules are generated for the efficient detection of malwares that exploits heap overflow vulnerability.

Rule#1: (TypeofStringz = HeapStrings) and (Countofloadedlibr <= 32) and (Numbercountofs <= 19) and (NumberofLoadedLibararies <= 0) => Overflow=HeapOverflow (25.0/1.0)

Rule#2: (TypeofMalware = Backdoor) and (TypeofStringz = HeapStrings) => Overflow=HeapOverflow (87.0/19.0)

Rule#3: (TypeofStringz = HeapStrings) and (Numbercountofs <= 11) and (NumberofMutex >= 1) and (NumberofMutex <= 16) => Overflow=HeapOverflow (32.0/9.0)

Rule#4: (NumberofProcessInteractions > 1) AND (TypeofStringz = HeapStrings) AND (TypeofMalware = Backdoor) AND (MutexName = Local\) AND (NumberofMutex <= 5) AND (NumberofProcessInteractions <= 5) AND (NumberofProcessInteractions > 4) => Overflow=HeapOverflow (47.0/6.0)

Rule#5: (LoadedLibraries = No) AND (Numbercountofs > 0) => HeapOverflow (23.0)

Rule#6: (TypeofStringz = HeapStrings) AND (MutexName = Local\) AND (NumberofProcessInteractions > 5) => HeapOverflow (16.0/1.0)

Rule#7: (TypeofStringz = HeapStrings) AND (Mutexcount > 2) AND (NetworkConnections <= 0) => HeapOverflow (11.0)

Rules#8: (TypeofStringz = HeapStrings) AND (MutexName = Global\) =>HeapOverflow (4.0)

Rule#9: (TypeofStringz = HeapStrings) AND (MutexName = Local\) AND (RegistryWrites = Yes) AND (RiskParameter = Autostart) AND (TypeofMalware = Backdoor) AND (NumberofMutex <= 4) AND (NumberofLoadedLibararies <= 102) AND (StigQueries = Yes) => HeapOverflow (4.0/1.0)

Using mining techniques to detect heap based overflow in a large dataset is much easier and takes less time as compared to conventional reverse engineering technique.

Chapter 8

Detection of Heap Based Overflow by using Classifiers

8.1 Introduction

Using classifiers to detect malwares that exploit heap based overflow vulnerability can be accomplished using a direct process. This process can take as little as few minutes or can be elongated to months, depending on the clarity of the objectives and scope, availability of dataset, and the pre-processing trials related with the data. Two rudiments of the analysis are collection of data and tool acquisition. The collected data entails a pre-processing stage to move it into the form which is required for classifier implantation and heap overflow detection. Result execution and analysis of data is a significant step to comprehend the subsequent model and its rule sets.

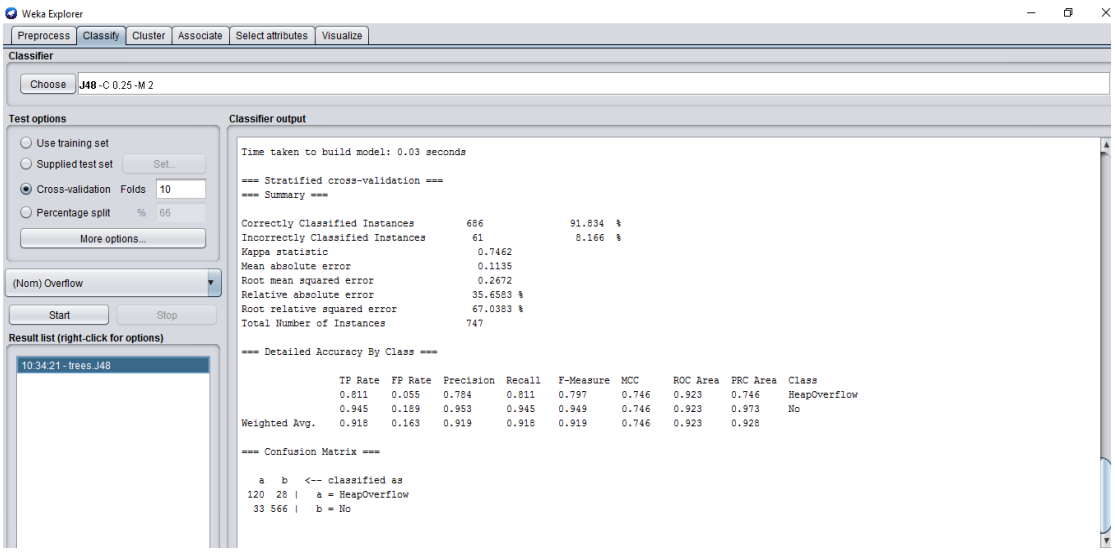
8.2 Result Execution and Analysis of Data

For the result execution and analysis an open source weka tool has been used. Weka is a best known data mining tool and provides a wide-ranging list of machine learning algorithms. The created dataset of 747 malware files comprising of 22 features are converted in to .arff format. ARFF files known as Attribute-Relation File Format used to work with weka machine learning software.

8.2.1 J48 Classifier

By implementing J48 classification algorithm we get an accuracy rate of 91.834%.

```
Time taken to build model: 0.03 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         686               91.834  %
Incorrectly Classified Instances        61                8.166  %
Kappa statistic                          0.7462
Mean absolute error                      0.1135
Root mean squared error                  0.2672
Relative absolute error                 35.6583 %
Root relative squared error             67.0383 %
Total Number of Instances              747
```



J48 tree detect the malwares which exploit heap based overflow vulnerability on the basis of following rules.

10:34:21 - trees.J48

```
LoadedLibraries  = Yes
|   NumberofProcessInteractions <= 4
|   |   TypeofStringz = HeapStrings
|   |   |   DNSQueries <= 0
|   |   |   |   Mutexcount <= 2
|   |   |   |   |   TypeofMalware = Adware
|   |   |   |   |   |   Mutexcount <= 0
|   |   |   |   |   |   |   RegistryWrites = Yes: HeapOverflow (3.0)
|   |   |   |   |   |   |   RegistryWrites = No: No (18.0/2.0)
|   |   |   |   |   |   Mutexcount > 0: HeapOverflow (14.0/3.0)
|   |   |   |   |   TypeofMalware = Backdoor: HeapOverflow (10.0/3.0)
|   |   |   |   |   TypeofMalware = Downloader: No (5.0/1.0)
|   |   |   |   |   TypeofMalware = Dropper
|   |   |   |   |   |   RiskParameter = Anomaly: No (0.0)
|   |   |   |   |   |   RiskParameter = Autostart: No (22.0)
|   |   |   |   |   |   RiskParameter = Disable: No (0.0)
|   |   |   |   |   |   RiskParameter = Evasion: HeapOverflow (2.0)
|   |   |   |   |   |   RiskParameter = Execution: No (0.0)
|   |   |   |   |   |   RiskParameter = File: No (0.0)
|   |   |   |   |   |   RiskParameter = Memory: No (0.0)
|   |   |   |   |   |   RiskParameter = Network: No (0.0)
|   |   |   |   |   |   RiskParameter = No: No (0.0)
|   |   |   |   |   |   RiskParameter = Packer: No (0.0)
|   |   |   |   |   |   RiskParameter = Reputation: No (0.0)
|   |   |   |   |   |   RiskParameter = Search: No (0.0)
|   |   |   |   |   |   RiskParameter = Signature: No (1.0)
|   |   |   |   |   TypeofMalware = Keylogger: No (29.0/5.0)
|   |   |   |   |   TypeofMalware = Rootkit: No (20.0)
|   |   |   |   |   TypeofMalware = Worm: No (10.0/1.0)
|   |   |   |   Mutexcount > 2
|   |   |   |   |   RiskParameter = Anomaly: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = Autostart: HeapOverflow (11.0/1.0)
|   |   |   |   |   RiskParameter = Disable: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = Evasion: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = Execution: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = File: No (4.0)
|   |   |   |   |   RiskParameter = Memory: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = Network: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = No: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = Packer: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = Reputation: HeapOverflow (0.0)
```

```
|   |   |   |   |   RiskParameter = Reputation: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = Search: HeapOverflow (0.0)
|   |   |   |   |   RiskParameter = Signature: HeapOverflow (1.0)
|   |   |   DNSQueries > 0: No (138.0/7.0)
|   |   TypeofStringz = No: No (149.0)
|   |   TypeofStringz = StackStrings: No (121.0)
|   NumberofProcessInteractions > 4
|   |   TypeofStringz = HeapStrings
|   |   |   RiskParameter = Anomaly: HeapOverflow (0.0)
|   |   |   RiskParameter = Autostart
|   |   |   |   NumberofMutex <= 6
|   |   |   |   |   NumberofMutex <= 5: HeapOverflow (70.0/10.0)
|   |   |   |   |   NumberofMutex > 5
|   |   |   |   |   |   NumberofProcessInteractions <= 6
|   |   |   |   |   |   |   Numbercountofs <= 296: No (6.0/1.0)
|   |   |   |   |   |   |   Numbercountofs > 296: HeapOverflow (3.0)
|   |   |   |   |   |   NumberofProcessInteractions > 6: HeapOverflow (7.0)
|   |   |   |   NumberofMutex > 6: No (21.0/3.0)
|   |   |   RiskParameter = Disable: HeapOverflow (0.0)
|   |   |   RiskParameter = Evasion: No (13.0)
|   |   |   RiskParameter = Execution: HeapOverflow (0.0)
|   |   |   RiskParameter = File: No (1.0)
|   |   |   RiskParameter = Memory: HeapOverflow (0.0)
|   |   |   RiskParameter = Network: HeapOverflow (0.0)
|   |   |   RiskParameter = No: HeapOverflow (0.0)
|   |   |   RiskParameter = Packer: HeapOverflow (0.0)
|   |   |   RiskParameter = Reputation: HeapOverflow (0.0)
|   |   |   RiskParameter = Search: HeapOverflow (0.0)
|   |   |   RiskParameter = Signature: HeapOverflow (0.0)
|   |   TypeofStringz = No: No (19.0)
|   |   TypeofStringz = StackStrings: No (23.0)
LoadedLibraries  = No
|   Numbercountofs <= 0: No (3.0/1.0)
|   Numbercountofs > 0: HeapOverflow (23.0)
```

Figure 22: Rules Generated by J48 Algorithm to detect Heap-Based Overflow

Visual representation of J48 tree is shown below.

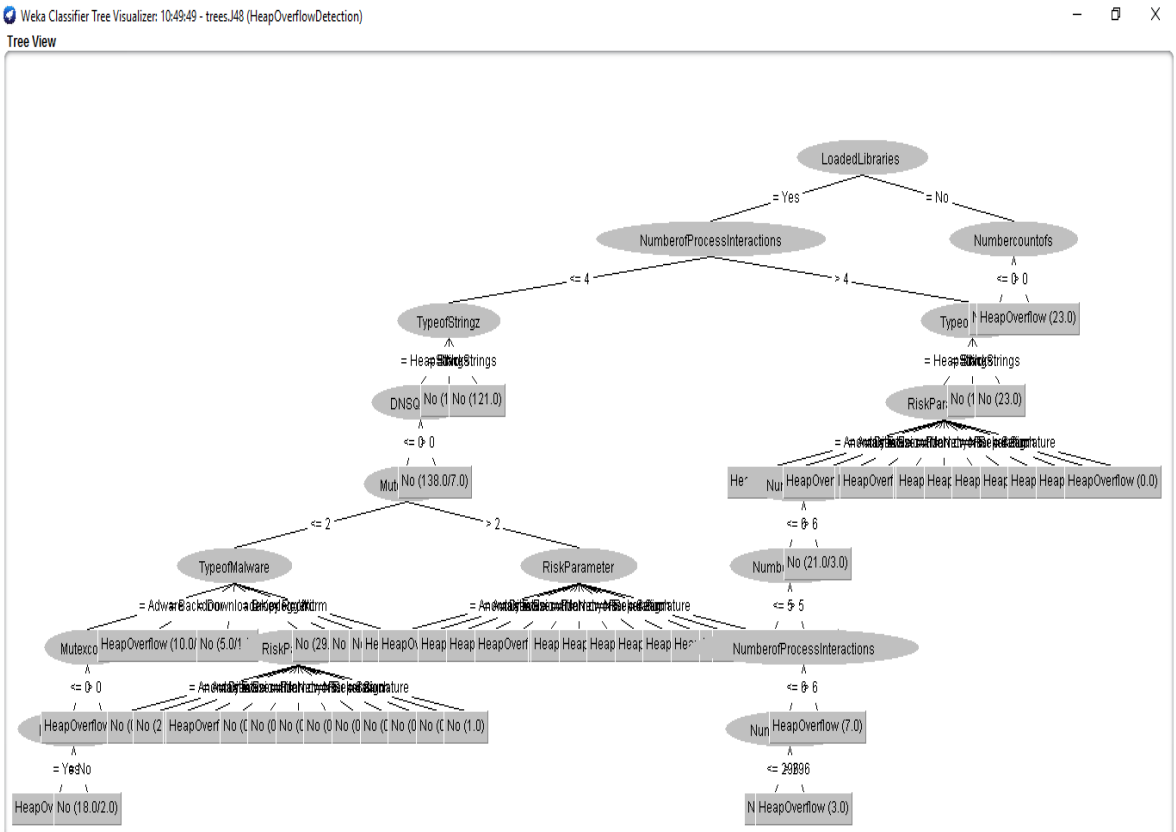
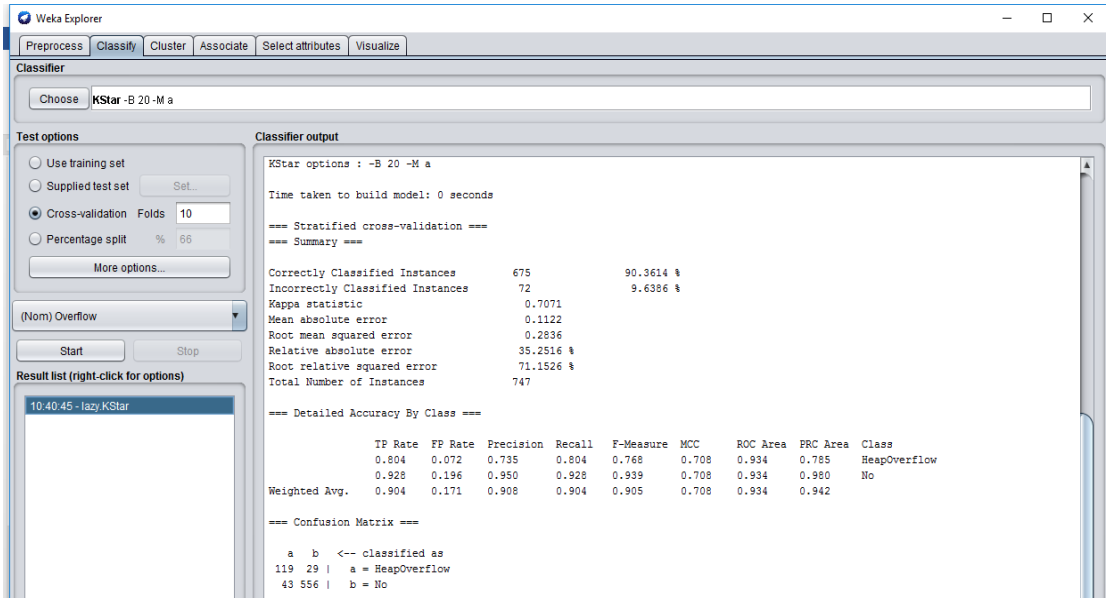


Figure 23: J48 tree

8.2.2 Lazy K-Start Classifier

By implementing Lazy K-Star classification algorithm we get an accuracy rate of 90.3614%.

```
Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         675               90.3614 %
Incorrectly Classified Instances        72                9.6386 %
Kappa statistic                          0.7071
Mean absolute error                      0.1122
Root mean squared error                  0.2836
Relative absolute error                 35.2516 %
Root relative squared error             71.1526 %
Total Number of Instances              747

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.804    0.072    0.735      0.804    0.768      0.708    0.934     0.785     HeapOverflow
                 0.928    0.196    0.950      0.928    0.939      0.708    0.934     0.980     No
Weighted Avg.    0.904    0.171    0.908      0.904    0.905      0.708    0.934     0.942
```

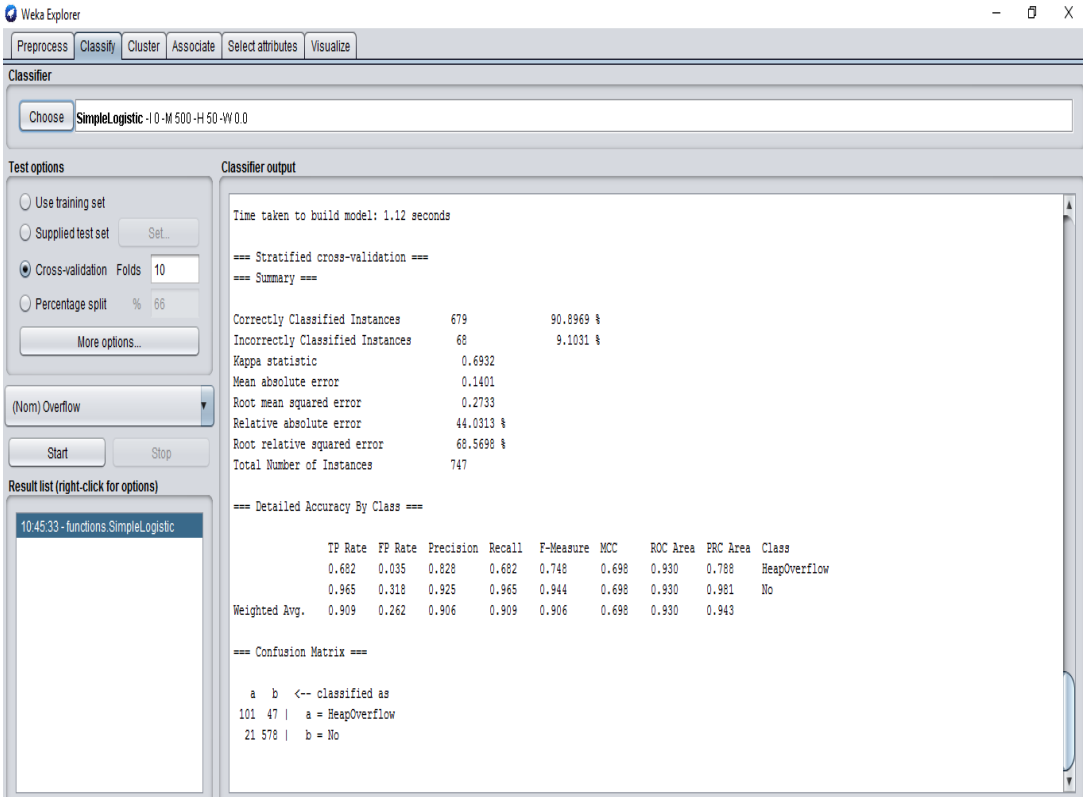


8.2.3 Simple Logistic

By implementing Simple Logistic classification algorithm we get an accuracy rate of 90.8969%.

```
Time taken to build model: 1.12 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         679               90.8969 %
Incorrectly Classified Instances        68                9.1031 %
Kappa statistic                          0.6932
Mean absolute error                      0.1401
Root mean squared error                  0.2733
Relative absolute error                 44.0313 %
Root relative squared error             68.5698 %
Total Number of Instances              747
```

```
=== Detailed Accuracy By Class ===

               TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
               0.682    0.035    0.828      0.682    0.748      0.698    0.930     0.788     HeapOverflow
               0.965    0.318    0.925      0.965    0.944      0.698    0.930     0.981     No
Weighted Avg.  0.909    0.262    0.906      0.909    0.906      0.698    0.930     0.943

=== Confusion Matrix ===

   a   b   <-- classified as
 101  47 |   a = HeapOverflow
  21 578 |   b = No
```



8.3 Conclusion

Implemented data mining classification algorithms are used to classify new portions of information into predefined groups. Classification algorithms use pre-classified dataset to classify data, based on current trends and patterns. After rule generation, the logic from the implemented algorithm can be combined into numerous intrusion detection technologies including firewalls and IDS signatures. Out of 3 implemented classifiers J48 algorithm shows highest accuracy rate of 91.834%.

Classification Algorithm	J48	K-Star	Simple Logistic
Accuracy Rate	91.834%	90.3614%	90.8969%

Chapter 9

Validation and Results

9.1 Introduction

To validate the performance of our proposed system 10-Fold cross validation is implemented. Cross validation computes the accuracy of the implemented model by dividing the dataset into training testing set. J48, K-Star and simple logistic classification models are created from the training set and their accuracy is calculated grounded on how well it classifies the testing set.

9.2 10-Fold Cross Validation

Three classifiers J48, KStar, Simple Logistic are trained and tested with 10 fold cross validation i.e., the created dataset is divided arbitrarily into 10 subsets, where 1 subset is used for testing and 9 for training. For every combination the process is repeated 10 times. This procedure aids in assessing the strength of a given approach to detect malwares that exploits heap based overflow vulnerability without any previous information.

For evaluation of the propose system the following quantities are considered:

- **True Positives (TP):** Number of malwares that exploit heap based overflow vulnerability are classified as malicious executable
- **False Positives (FP):** Number of benign programs classified as malicious malware that exploit heap based overflow vulnerability

Classification Algorithm	J48	K-Star	Simple Logistic
Accuracy Rate	91.834%	90.3614%	90.8969%
True Positives (TP)	0.918	0.904	0.909
False Positives (FP)	0.163	0.171	0.262

Table 3: Accuracy Rate of Classifiers

Implementation of 10-Fold Cross Validation technique has significantly increase the accuracy rate of the system. The comparison of the modeled data is given below

Classification Algorithm	J48	K-Star	Simple Logistic
Accuracy Rate (10-Fold Cross Validation)	91.834%	90.3614%	90.8969%
Accuracy Rate (Percentage Split)	88.5827%	89.7638%	88.189%

Table 4: Comparison of Accuracy rate before and after Data Modeling

9.3 Conclusion

By the implementation of proposed system a comprehensive detection technique is presented for classification of malwares that exploits heap based overflow vulnerability.

The logic from the implemented algorithm could be used to upsurge awareness of APT strategies, and advance the complete security of the organizations. Details of our work is concluded in this chapter and future work is discussed in detail in next chapter.

Chapter 10

Future Work

10.1 Introduction

Polymorphic variants and generation of new malware families are forcing the Anti-Malware industry to create automatic tools to classify the malwares ability to exploit vulnerabilities and their corresponding class. In our proposed research work we have presented a behavioral detection technique for the malwares exploiting heap overflow vulnerability. In this chapter, our research work is concluded.

10.2 Future Work

The proposed methodology can be used by security analysts to protect systems from memory manipulation errors. It can be used for the behavioral analysis of malwares to detect APT treats. This approach can be used to detect Memory Leaks, Stack Overflow, and Integer Overflow. This approach can be combined with Address Space Layout Randomization (ASLR) technology to efficiently detect and prevent heap based overflow malwares.

10.3 Conclusion

Our work presents a malware detection system using combination of data mining and reverse engineering techniques. The proposed system is based on mining features of the binary files for detecting malwares that can exploit heap based overflow vulnerability. We developed a Feature parser to parse features from malware files and store them in to a dataset. A labeled dataset is created representing the class of malwares and their ability to exploit heap based overflow vulnerability. The created feature set is used to train three classifiers J48, K-Star and Simple logistic for the detection of heap based overflow. The proposed methodology is easy to implement

in operations of cyber security to comprehend the behavior of malwares targeting their organizations.

References

[1] Anitta Patience Namanya et al. “Detection of Malicious Portable Executables Using Evidence Combinational Theory with Fuzzy Hashing.” In Proc. Future Internet of Things and Cloud (FiCloud), 2016 IEEE 4th International Conference on, 2016

[2] Maryam Mouzarani et al. “Smart fuzzing method for detecting stack-based buffer overflow in binary codes”. IET Software, vol. 10, pp. 96-107, 2016

[3] Maryam Mouzarani et al. “A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes.” In *Proc. Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on,* 2015

[4] V.K.Gudipati et al. “Detection of Trojan Horses by the analysis of system behavior and data packets.” In *Proc. Systems, Applications and Technology Conference (LISAT), 2015 IEEE Long Island,* 2015

[5] G.G.Sundarkumar et al. “Malware detection via API calls, topic models and machine learning.” In *Proc. Automation Science and Engineering (CASE), 2015 IEEE International Conference on,* 2015

[6] Muhammad N. Sakib et al. “Automated Collection and Analysis of Malware Disseminated via Online Advertising.” In *Proc. Trustcom/BigDataSE/ISPA, 2015 IEEE*, 2015

[7] Zhenyi Liao et al. “A stack-based lightweight approach to detect kernel-level rookits.” In *Proc. Progress in Informatics and Computing (PIC), 2015 IEEE International Conference on,* 2015

[8] Zane Markel and Michael Bilzor. “Building a machine learning classifier for malware detection.” In *Proc. Anti-malware Testing Research (WATeR), 2014 Second Workshop on,* 2014

[9] Nir Nissim et al. “ALPD: Active Learning Framework for Enhancing the Detection of Malicious PDF Files.” In *Proc. Intelligence and Security Informatics Conference (JISIC), 2014 IEEE Joint,* 2014

[10] Sandeep Kumar et al. “Malicious data classification using structural information and behavioral specifications in executables.” In Proc. *Engineering and Computational Sciences (RAECS), 2014 Recent Advances in,* 2014

[11] S.K. Pandey et al. “Performance of malware detection tools: A comparison.” In *Proc. Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on,* 2014

[12] Mikhail Zolotukhin et al. “Detection of zero-day malware based on the analysis of opcode sequences.” In *Proc. Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th,* 2014

[13] U. Baldangombo, "A static malware detection system using data mining methods," (IJAIA), Vol. 4, No. , July 2013

[14] Szekeres, L., Payer, M., Wei, T., et al.: ‘Sok: Eternal war in memory’. Proc. 2013 IEEE Symp. on Security and Privacy (SP), 2013, pp. 48–62