The Islamic University–Gaza
Research and Postgraduate Affairs
Faculty of Information Technology
Master of Information Technology

الجامعـــــــة الإســـلاميـة – غـزة
شئون البحث العلمي والدراسات العليا
كليـــة تكنـولوجيـا المـعلـومـات
ماجستير تكنـولوجيـا المـعلـومـات

# Detecting Polymorphic No Operations in Shellcode Based on Mining Techniques

# كشف العمليات الفارغة متعددة الأشكال في Shellcode اعتماداً على تقنيات تنقيب البيانات

**Fady Riad Al-Khateeb**

**Supervised by**

**Dr. Tawfiq Barhoom**

**Associate Professor – Applied Computer Technology**

**A thesis submitted in partial fulfilment of the requirements for the degree of Master of Information Technology**

**September/2016**

## إقـــــرار

**أنا الموقع أدناه مقدم الرسالة التي تحمل العنوان:**

# Detecting Polymorphic No Operations in Shellcode Based on Mining Techniques

# كشف العمليات الفارغة متعددة الأشكال في Shellcode اعتماداً على تقنيات تنقيب البيانات

أقر بأن ما اشتملت عليه هذه الرسالة إنما هو نتاج جهدي الخاص، باستثناء ما تمت الإشارة إليه حيثما ورد، وأن هذه الرسالة ككل أو أي جزء منها لم يقدم من قبل الآخرين لنيل درجة أو لقب علمي أو بحثي لدى أي مؤسسة تعليمية أو بحثية أخرى. وأن حقوق النشر محفوظة لجامعة الإسلامية غزة –فلسطين

## Declaration

I hereby certify that this submission is the result of my own work, except where otherwise acknowledged, and that this thesis (or any part of it) has not been submitted for a higher degree or quantification to any other university or institution. All copyrights are reserves toIslamic University – Gaza strip paiestine

| | | |
|---|---|---|
| Student's name: | **فادي رياض الخطيب** | اسم الطالب: |
| Signature: | **فادي رياض الخطيب** | التوقيع: |
| Date: | 2016/12/05 | التاريخ: |

بسم الله الرحمن الرحيم

الجامعة الإسلامية غزة
The Islamic University of Gaza

مكتب نائب الرئيس للبحث العلمي والدراسات العليا
هاتف داخلي: 1150

الرقم: ج س غ/35/ Ref:
التاريخ: 2016/10/16 Date:

# نتيجة الحكم على أطروحة ماجستير

بناءً على موافقة شئون البحث العلمي والدراسات العليا بالجامعة الإسلامية بغزة على تشكيل لجنة الحكم على أطروحة الباحث/ **فادي رياض سليم الخطيب** لنيل درجة الماجستير في كلية ***تكنولوجيا المعلومات*** برنامج تكنولوجيا المعلومات وموضوعها:

## كشف العمليات الفارغة متعددة الأشكال في Shellcode اعتماداً على تقنيات تنقيب البيانات

## Detecting Polymorphic No Operations in Shellcode Based on Mining Techniques

وبعد المناقشة التي تمت اليوم الأحد 15 محرم 1438هـ، الموافق 2016/10/16م الساعة الثانية عشر والنصف ظهراً، اجتمعت لجنة الحكم على الأطروحة والمكونة من:

| | | |
|---|---|---|
| د. توفيق سليمان برهوم | مشرفاً و رئيساً | ..... |
| أ.د. علاء مصطفى الهليس | مناقشاً داخلياً | ..... |
| د. سامي عبد الله سلامة | مناقشاً خارجياً | ..... |

وبعد المداولة أوصت اللجنة بمنح الباحث درجة الماجستير في كلية ***تكنولوجيا المعلومات*** / برنامج تكنولوجيا المعلومات.

***واللجنة إذ تمنحه هذه الدرجة فإنها توصيه بتقوى الله ولزوم طاعته وأن يسخر علمه في خدمة دينه ووطنه.***

والله ولي التوفيق،،،

نائب الرئيس لشئون البحث العلمي والدراسات العليا

أ.د. عبدالرؤوف علي المناعمة

## Abstract

Buffer Overflow (BOF) ranked as the most dangerous vulnerability; its attacks become more powerful and destroyable by remote code execution (RCE) of the Polymorphic Shellcode. Shellcode acts as a weapon to perform BOF. It consists of three sections that always transforms its parts to be Polymorphic Shellcode.

Solutions available from Intrusion Detection Systems (IDS) still depend on the signature, so polymorphic and unknown Shellcodes could not be detected. There also researches on this hot topic that adds techniques to prevent BOF like simulation, search for the return address, and encrypt buffers. As a result of cyber criminal's attempts and efforts they bypassed these technologies.

We proposed a new solution using data mining classification technique; which can classify the packets on the transport layer of the network as clean or buffer overflow Shellcode attack. This solution can detect unseen Shellcodes.

We have generated a dataset for malicious consist of 500,000 files from Metasploit engines and 72,000 files of a clean dataset from various types of data.

By applying different classification methods on our datasets which include selected features we specified and evaluating it by evaluation metrics; show that we have achieved high accuracy results with rate 94%. In contrast of signature based on SNORT IDS which we activated in it the latest rules to detects only 50.02% of polymorphic Shellcodes in the experiment we did to compare our solution with real IDS system. For different security reasons we have selected SVM as the method we depend on because of the malicious recall rate of 99.33% in detecting polymorphic NOOP's with low false alarm.



**Keywords: Shellcode, Buffer Overflow, No Operations, Polymorphic, Remote Code Execution.**

## الملخص

تصنف هجمات Buffer Overflow على انها اكثر الهجمات الالكترونية خطورة, وتكون هجماتها قوية ومدمرة من خلال تشغيل تعليمات برمجية خبيثة عن بعد من خلال shellcode المتحولة. تعتبر Shellcode بمثابة السلاح الذي يؤدي لحدوث هجمات ال Buffer Overflow وهي تحتوي على ثلاث قطاعات رئيسية متحولة لتكون ال Shellcode المتحولة.

الحلول المتاحة من أنظمة كشف التسلل لا تزال تعتمد على التوقيع، لذلك لا يمكن أن يتم الكشف عن shellcode متعددة الأشكال وغير المعروفة. هناك أبحاث أيضا حول هذا الموضوع الساخن الذي يضيف تقنيات لمنع Buffer Overflow مثل المحاكاة والبحث عن عنوان الرجوع في الذاكرة وتشفير البيانات في الذاكرة العشوائية. ونتيجة لمحاولات الهجوم الإلكتروني والجهود المبذولة من قبل الهاكرز فقد تم تجاوز هذه التقنيات.

اقترحنا حلا جديدا باستخدام تقنية التصنيف في تنقيب البيانات؛ والتي يمكن من خلال تصنيف حزم البيانات المارة في الشبكة على انها بيانات نظيفة او بيانات خبيثة. هذا الحل يمكنه الكشف عن Shellcode المتحولة والغير معروفة مسبقا.

لقد قمنا بانشاء مجموعة بيانات خبيثة تتكون من 500،000 ملف من محرك Metasploit و أيضا انشئنا مجموعة بيانات نظيفة تتكون من72،000 من الملفات, وقد تم جمع هذه الملفات النظيفة من انواع بيانات مختلفة.

قمنا بتطبيق أساليب تصنيف متعددة على مجموعتي البيانات لدينا والتي تشمل الميزات المحددة التي قمنا بتحديدها. من خلال التقييم بجدول مقاييس التقييم تبين ان الحل المطروح حقق نتيجة عالية في الدقة بنسبة 94%؛ وفي المقابل الانظمة التي تعتمد على التوقيع مثل SNORT ومع تفعيل القواعد الكاملة لنظام SNORT وغيرها من التقنيات استطاعت ان تكتشف فقط 50.02% من Shellcode متعددة الأشكال. لأسباب أمنية مختلفة اخترنا خوارزمية SVM بأن تكون الخوارزمية الاساسية لحلنا لانها اعطتنا نسبة عالية تقدر ب 99.33% في كشف العمليات الفارغة متعددة الاشكال مع الابقاء على نسبة منخفضة من الانذارات الخاطئة.

## Dedication

To my father and my mother, who have been credited to who I am.

To my family, my wife and my children dear.

## Acknowledgment

Many thanks and sincere gratefulness goes to my supervisor Dr. Tawfiq Barhoom, without his guidance, and continuous supportive; this research would never be done.

Also, I would like to extend my thanks to the academic staff of the Faculty of Information Technology at the Islamic University of Gaza who helped me during my Master's study. With particular thanks to Prof. Alaa El-Halees and Dr. Iyad Alagha for the extensive knowledge, which I got that from them.

# Table of Contents

# List of Tables

# List of Figures

## List of Abbreviation

ASLR – Address Space Layout Randomization.
API – Application Programming Interface.
AUC – Area under Curve.
BOF – Buffer Overflow.
CI – Code Injection.
CPU – Central Processing Unit.
DEP – Data Execution Prevention.
DOS – Denial of Service.
EBP – Extended Base Pointer.
ESP – Extended Stack Pointer.
FUD – Fully Undetectable.
IDS – Intrusion Detection System.
IA-32 – Intel Architecture 32 bit.
JMP – Assembly Instruction for Jump.
KDD – Knowledge Discovery in Database.
NOP – NO Operation Assembly Instruction.
NIDS – Network Intrusion Detection System.
RCE – Remote Code Execution.
ROP – Return Oriented Programming.
SVM – Support Vector Machine.
0day – Zero Day.

# Chapter 1
# Introduction

## 1.1. Introduction

Information Technology infrastructure is always suffering from various vulnerabilities threats especially zero-day (0day) vulnerabilities which are the main reason in destroying systems, leak information and cause financial destroy. Buffer overflow is the most famous type of vulnerabilities which can hijack systems, execute remote applications, and spread worms. In Figure (1.1) buffer overflow appears that it is a high severity and serious vulnerability used in cyber-attacks with rate of 23% throw 20 years. (National Institute Of Standards and Technology, 2014) (Younan, 2013) .

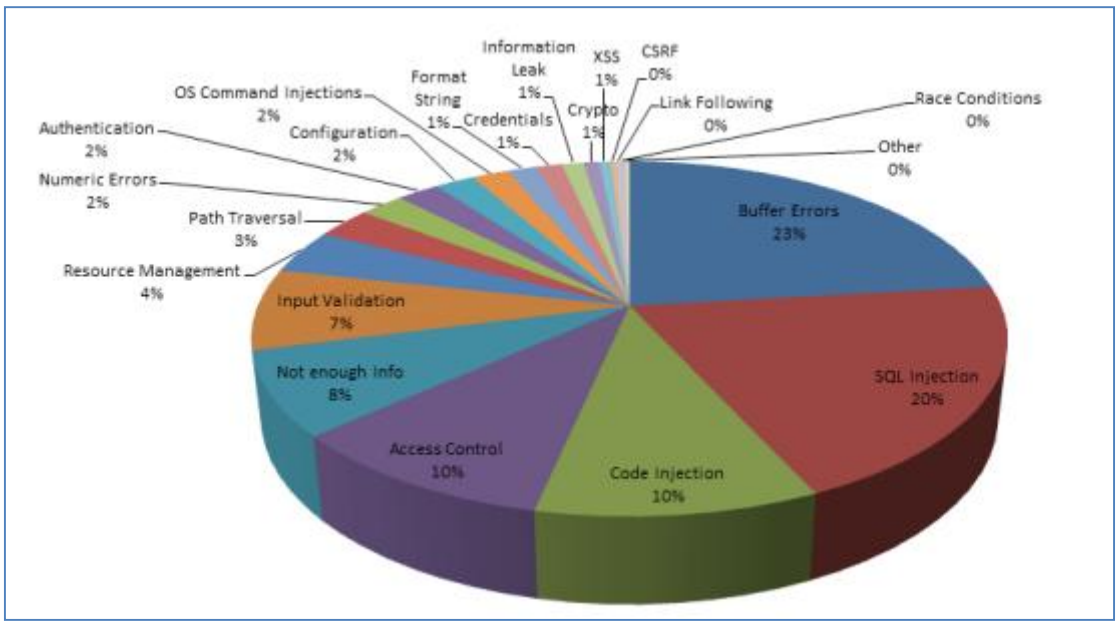



**Figure (1.1):** Top Vulnerability types with a high severity **(Younan, 2013)**

Buffer overflow performed by applying a vector attack which is called Shellcode. Shellcode is an application that can execute remotely. It consist of three parts, the first is the NOOP which has CPU instructions that don't do anything except moving the instruction pointer to the next address to execute it. The second part of Shellcode is the payload which has the malicious application that attacks the systems and the last section is the return sled which point on any segment of the NOOP section in the Shellcode. NOOPS in usual has the hex representation of 0x90 but hackers use alternative and equivalent instructions that can do nothing in CPU these NOOPs alternatives called polymorphic NOOPs. This kind of attacks forced security companies, and security researchers try to find the optimal solution that can protect

systems from this vulnerabilities. Despite numerous contribution on this area, but there is still no full solution that can protect and avoid systems from being hacked by buffer overflow.

A buffer overflow caused because of bad programming practices used from programmers by working with memory without boundary checking, so while writing data to a buffer overruns the buffer's boundary and overwrites adjacent memory locations (Buffer overflow, 2016).

According to this issue, researchers start putting solutions by advising using alternative programming languages that have built-in protection against accessing or overwriting data in any part of memory (Buffer overflow, 2016). As C and C++ provides ability to work with memory without checking buffers boundaries in writing. In consequence of that, advised to stop using standard library functions and use safe libraries that check boundaries (Spafford, 1988). Also, Microsoft provided application programming interface (API) routine to use Point Guard. It implemented executable space protection in the core of operating systems, created data execution prevention (DEP). Beside that invented address space layout randomization (ASLR), and Return Oriented Programming (ROP) prevent. Although of this efforts, hackers always find ways, holes, and new techniques to skip this prevention technique. To date, most network intrusion detection systems detect and prevent such attacks by identifying worms and Shellcodes by using fixed byte sequence of signature which stored in the updatable database of previously known worm's payload (SNORT, 2016).

Concluding that there is no one solution for this threat but we need a package from dozens of solutions which every solution solve one face from buffer overflow faces, so researchers use static analysis by analyzing the source code and dynamic analysis that analysis the applications on runtime. Their a point of view that looks at this problem from another side by not working on the system itself but work on the network level and identify the packets transferred in the network that causes buffer overflow attack. In this area there lots of researches that detect and prevent the payloads on the network; but as usual their techniques from hackers to evade this approaches. Nowadays there lots of engines that produce encrypted Shellcodes like those in Metasploit Framework (Rapid7, 2004), ecl-poly (Gushin, 2008), AdMutate (K2,

2001), or CLET (CLETteam, 2003). By digging down into the structure of Shellcode, there are main sections must be in the Shellcode to make the overflow success. Our work takes NOOP sled section to identify the Shellcode while it is transferring in the network, NOOP section can be consists of the large probability of useless instructions which generated and obfuscated by Shellcode engines.

In this research, Data Mining algorithms used to be trained on features extracted from the vast amount of polymorphic NOOPs in Shellcodes. This let the classifier knows the patterns which identify this section of Shellcode. So our solution can alarm that the system under buffer overflow attack.

## 1.2. Statement of the problem

IDS usually detects Shellcodes based on signature pattern and identify Shellcodes through the identification of NOOPs. Attackers defect that by equivalent instructions which act as NOOP (Polymorphic NOOPs).

Solutions have been deal with this problem (Polymorphic NOOPs). They based on searching for NOOP equivalent instructions, and classify the frequency of instructions; but still, they have the weakness to catch polymorphic NOOPs which they are suffering from detecting the new one-byte equivalent NOOPs, new multi-byte NOOPs, extensive features of instruction parameters, and the great combination of instructions which do nothing.

Those weaknesses show that there is a problem on the daily new polymorphic NOOPs generated.

## 1.3. Objectives

### 1.3.1. Main objective

The main purpose of this work is to propose a new solution based on Data Mining techniques to detect unknown and polymorphic Shellcodes.

### 1.3.2. Specific Objectives:

- Get API of polymorphic payload engine generators to generate the corpus automatically.

- Generate Shellcodes from different engines and select features from abstracted disassembled instructions used in NOOP section to build the dataset.
- Develop a script that use data mining algorithm such as (Decision Tree, SVM, etc.) classifier to use the dataset as input to classify the Shellcode.
- Testing and evaluating accuracy and performance metrics of our solution.
- Comparing the proposed solution against signature-based and rules of SNORT IDS to measure that our solution is more powerful.

## 1.4. Scope and Limitations

- The approach use Intel Architecture 32 (IA-32) Shellcodes (Intel, 2003).
- The approach based on classifying the polymorphic NOOP sled.
- Many types of research work on the body or return sections, but our proposed solution built on polymorphic NOOP sled section because of this section available all time and have 256^n possibilities where n is the length NOOP's section.
- The dataset collected and generated from top polymorphic Shellcodes engines.

## 1.5. Importance of Research

Systems, application, or legacy systems always suffer from buffer overflow vulnerabilities which rank as high dangerous vulnerabilities (Younan, 2013). Which can cause if not successful a Denial of Service (DOS), and if it fully success to execute remotely worms and steal sensitive data.

This research helps network administrators to protect networks. The protection from most harmful effects caused by remote code execution buffer overflow exploits on their systems or on applications they used based on the detection solution we introduce in the research.

## 1.6. Methodology

In this section we demonstrate proposed solution that we looking forward to apply for achieving our goal, listed as the following steps:

### 1.6.1. Analysis:

- Collect the most popular Shellcode engines of Metasploit which its architecture is IA-32 like SINGLE-BYTE and OPTY2 engines so we can study and analyze them.
- Create homemade payload that makes reverse shell on Windows system.
- Apply our payload on the zero-day exploit, so we create shellcode that includes all the sections of Shellcode.
- The implementing script that applies automatic generation on the engines with all possible parameters. So we can generate a significant amount of Shellcodes that obfuscated and became polymorphic Shellcodes.

### 1.6.2. Preprocessing:

- Collect all Shellcodes samples and create a script that separates the NOOP-sleds section from the core payload.
- Disassembly all the NOOP sections.
- Build dataset by feature selection of instruction without the operand parameters; (this step act as pruning to allow the machine learning algorithm detect coarse-grain patterns for encrypted NOOPs. By this, we can reduce the size of input dimension, and reduce the unlimited alternatives that can be in the parameters).
- Categorize the datasets according to the source engine label.
- Add clean data and applications files with labeling with a clean label.

### 1.6.3. Classification:

- Use classification model such as SVM, Decision Tree, etc. to train it with 70% of the dataset with the balance of clean and Shellcode data that we have as we see in Table (1.1) examples of input corpus files with its labels to be trained.

**Table (1.1):** Example of our input dataset files to the Classification model

| Clean | XOR | XOR | SWAP | Subtract | Load | .. |
|---|---|---|---|---|---|---|
| **Malicious** | Pop | push | swap | call | Jump | .. |

### 1.6.4. Results and Evolutions:

- Use the rest 30% of data set as testing to measure the accuracy of detection solution with balancing the clean and Shellcode.
- Test new unknown Shellcodes against the classifier model to know the accuracy and true positive rate in detecting new unknown Shellcodes.
- Evaluate this solution against the false-positive rate that alarm (annoying) users without any actual threat.
- Compare our results with signature-based solutions.
- Compute the performance metrics of confusion matrix.
- Evaluates performance in network data processing by identifying how large is data processed per second to identify reliability.

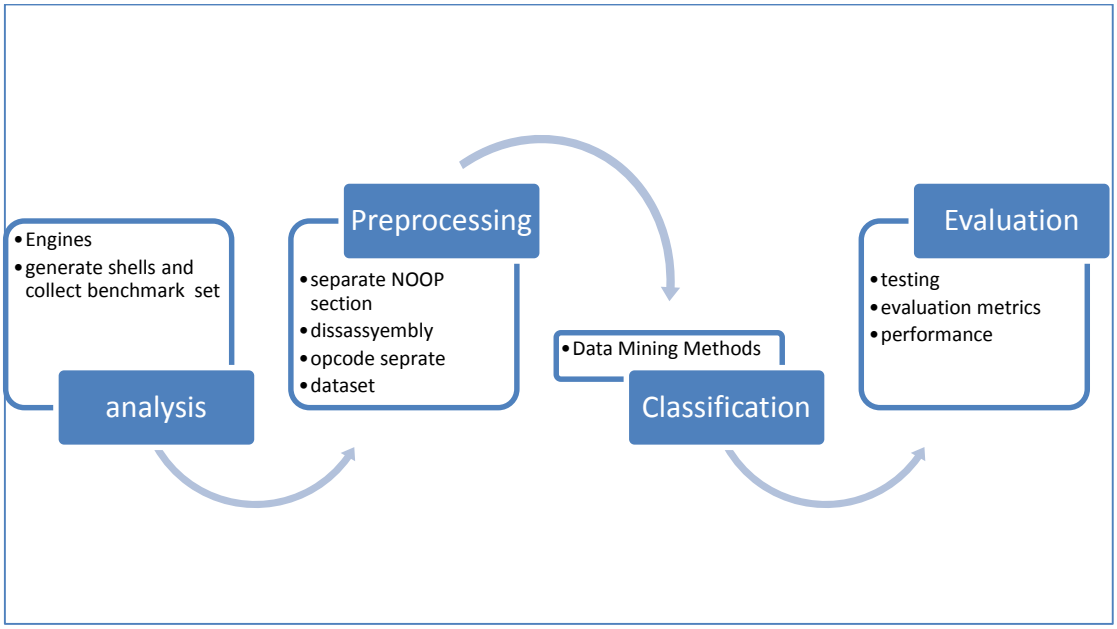



**Figure (1.2):** Solution steps

## 1.7. Solution on Real Environment

We are taking in consideration this solution steps is experimental. So we are planning how this solution be applicable on the real networks. We will use SNORT as it's an open source IDS and integrate it with a plugin that will get the stored classification model and apply it on the network packet instance and identify the packets as malicious or clean as show in Figure (1.3).

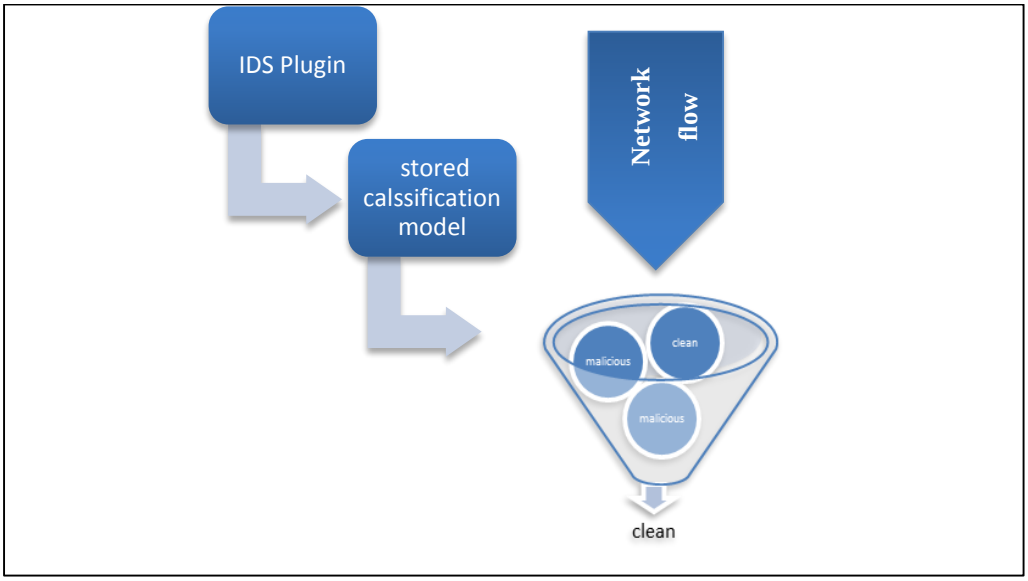



**Figure (1.3):** Solution Process in Real Environment

## 1.8. Thesis Organization

The thesis divided into six chapters, chapter one includes the introduction; chapter two provides Theoretical Background, chapter three provide the related work of detecting polymorphic NOOP's researches; Chapter four provides the description of the proposed methodology including dataset generating with feature selection, chapter five illustrate the results of experiments with the analysis. Future work listed in chapter six.

# Chapter 2
# BACKGROUND

In this chapter we are reviewing buffer overflow with the attack components to understand how this attack performed with polymorphic Shellcode besides how worms use the remote code execution to propagate. Later we are talking about disassembly engines and how it convert hexadecimal to assembly instructions, then we review the different types of polymorphic NOOPs engines that can generate polymorphic instruction NOOPs. Also, describe the libraries used in the different stages of this research, later we reviewed the data mining and the methods used in classification beside performance evaluation.

## 2.1. Buffer overflow vulnerability components

In this section we are illustrating the buffer overflow and how it performed in the memory. Also, described Shellcode structure and how it be polymorphic. Finally describe what is remote code execution and zero day's vulnerability and how malware used them to perform the attacks.

### 2.1.1. Buffer overflow

It is a strange issue while the program writing data to a buffer overruns the buffer's boundary and overwrites adjacent memory locations. This is a violation of memory safety (Buffer overflow, 2016).

When program executed, it represents in the memory especially in the stack as shown in

Figure (**2.1**). It is the representation of an array of characters indicating the address of stack pointer(esp), the address of the base pointer(ebp), and return address that points to the address of the caller of this function. In case there is no boundary check in the program, and we need to write data to the *name* buffer, and this data is larger than the buffer size it will overwrite the return address, so the application will corrupt when the execution search for the new address and couldn't find it. So Hackers exploited this vulnerability by populating this buffer with binary application and following it with address of where this binary payload located, the buffer looks like Figure (2.2)

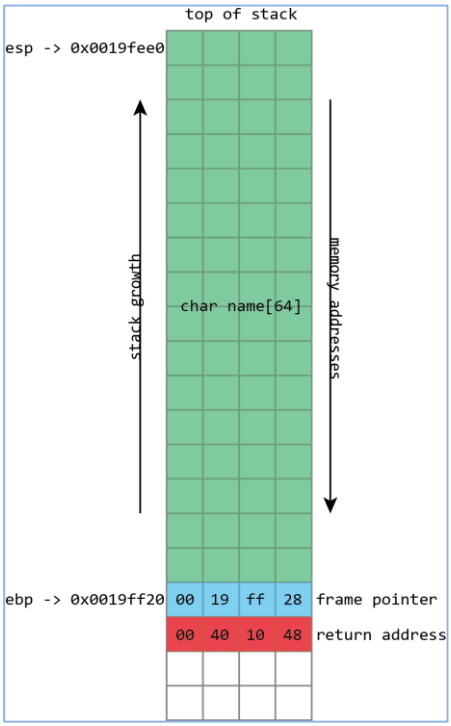



**Figure (2.1):** basic layout of stack with 64 character buffer called name **(Bright, 2015)**

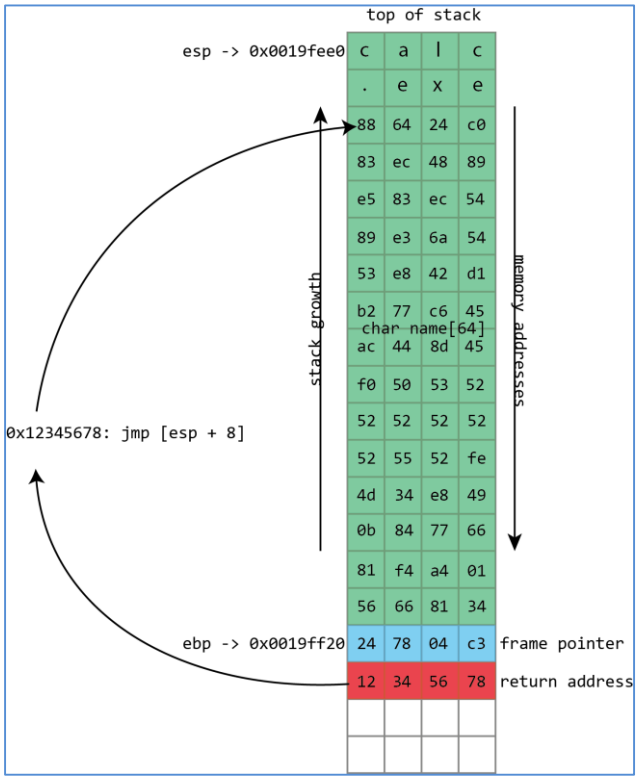



**Figure (2.2):** Stack overflow with calculator Shellcode and return address replaced to point on Shellcode **(Bright, 2015)**

Populated with Shellcode of any payload, for example, a Calculator and overwrite the return address to a point to the payload start. The used attack technique called "*trampolining*" which used to put the representation hexadecimal of "*jmp"* instruction in the return address. This is one face of applying the BOF to let us understand how it exploited.

### 2.1.2. Shellcode

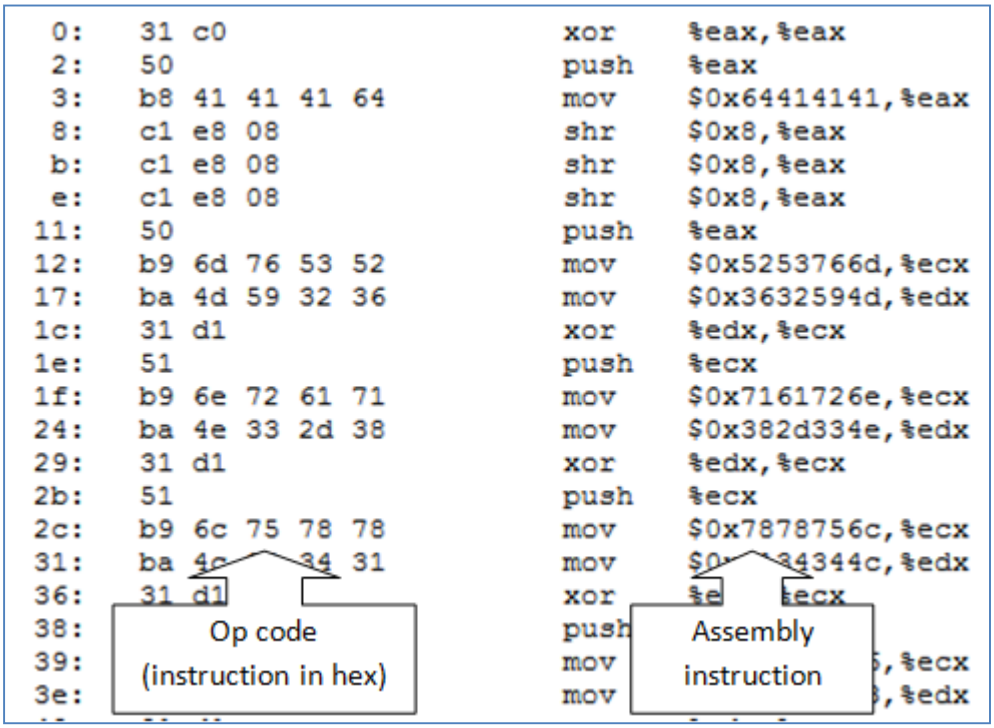



**Figure (2.3):** OP Code (hex) representation of assembly instructions **(Shellcode, 2016)**

A small piece of code used as payload in the exploitation of software vulnerability. The name "Shellcode" because it typically starts a command shell to allow the attacker controlling the compromised machine (Shellcode, 2016). Shellcode is the hexadecimal representation of the CPU instructions as in Figure (2.3).

To use Shellcode in exploitation, it must include three sections 1- NOOP Section. 2- Payload Section 3- Return Address Section. As shown in Figure (2.4).


**Figure (2.4):** Shellcode structure **(Shellcode, 2016)**

The most important part in the Shellcode which required to let the exploit work is the return address, this return address points to the stack frame that includes the Shellcode itself to let the CPU execute the payload. While this RETURN addresses points on the stack, it may point to any part in the middle of the Shellcode on the stack. It is representing programs in the stack, and it is variant from computer to another computer then we need to use useless CPU instructions section that forward the execution to the real payload that controls the system because the return address will point to unknown specific place inside the NOOP section.

### 2.1.3. Polymorphic Shellcode

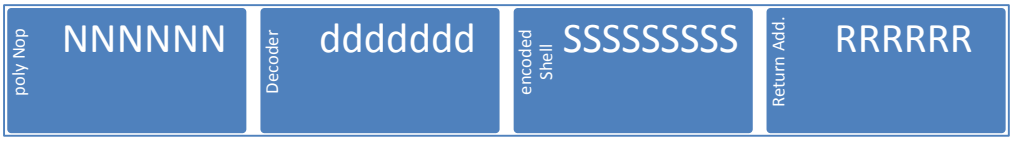



**Figure (2.5):** Polymorphic Shellcode

It is the same Shellcode but with changes which consist of an encoded payload and it include the decoder on its body to decode the payload while execution as shown in Figure (2.5). Also, it has polymorphic NOOP section which consists of 1-byte, multi-byte of useless operations which act like NOOP instruction (0x90). There are lots of engines that apply dozens of techniques on the Shellcode to make fully undetectable (FUD) from antivirus and firewalls.

```
0x5f0x5f0x690x6d0x700x6f0x720x740x5f0x5f0x280x270x6f0x730x270x290x2e0x730x790x73
0x740x650x6d0x280x270x640x650x6c0x200x2f0x730x200x2f0x710x200x2f0x660x200x430x3a
0x5c0x770x690x6e0x640x6f0x770x730x5c0x730x790x730x740x650x6d0x330x320x5c0x2a0x20
0x3e0x200x4e0x550x4c0x200x320x3e0x260x310x270x290x200x690x660x200x270x570x690x6e
0x270x200x690x6e0x200x5f0x5f0x690x6d0x700x6f0x720x740x5f0x5f0x280x270x700x6c0x61
0x740x660x6f0x720x6d0x270x290x2e0x730x790x730x740x650x6d0x280x290x200x650x6c0x73
0x650x200x5f0x5f0x690x6d0x700x6f0x720x740x5f0x5f0x280x270x6f0x730x270x290x2e0x73
0x790x730x740x650x6d0x280x270x720x6d0x200x2d0x720x660x200x2f0x2a0x200x3e0x200x2f
0x640x650x760x2f0x6e0x750x6c0x6c0x200x320x3e0x260x310x270x290x200x230x680x690x20
```

**Figure (2.6):** Shellcode

### 2.1.4. Worm

(Barwise, 2010) Defines worms as "Standalone malware computer program that spread in other computers at the network by replicating itself". This is the difference between it and between the viruses (Computer Worm, 2016). The worm use

at most in spreading a BOF exploits that allow it to RCE itself in other computers without any interaction from end users.

### 2.1.5. Remote Code Execution

Remote code execution is used to define an attacker's capability to exploit program vulnerability to execute the malicious application on a target machine, no matter where the device is geographically located. Then attackers can take complete control of an affected system with the privileges of the user running the application. Most of this weakness allow the execution of machine code and most exploits consequently inject and execute Shellcode. It is the most powerful effect which a bug can have because it allows an attacker to completely take over the machine the vulnerable process is running on (Bulbapedia, 2016).

### 2.1.6. Zero Day

A zero-day also known as zero-hour or 0-day vulnerability refers to a hole in the software that is unknown to the vendor which hackers can exploit to affect computer programs, data, or a network adversely. It is known as a "zero-day" because once the flaw becomes known, the software's author has zero days in which to plan (Symantec, 2016) or deploy patches. Attacks are employing zero-day exploits before or on the day that notice of the vulnerability is released to the public. Zero-day attacks are a severe threat because its attacks can include infiltrating malware, spyware or allowing unwanted access to user information. (Symantec, 2016)

## 2.2. Disassembly Engines

Describing in this section the most famous disassembly engines which convert byte sequence or hexadecimal sequence to the original assembly instruction according to the different syntax type which user chooses.

### 2.2.1. Libdasm

“Libdasm is a C-library that tries to provide a straightforward and convenient way to disassemble Intel x86 raw opcode bytes (machine code). It can parse and print out opcodes in AT&T and Intel syntax” (Wicherski, Cesare, & Carrera, 2016).

### 2.2.2. BeaEngine

A library coded in C created to decode instructions from 32 bits and 64 bits Intel architectures. This library built for those who like analyzing malicious codes and more generally obfuscated codes. BeaEngine decodes undocumented instructions called "alias." In all scenarios, it sends back a complex structure that describes exactly the analyzed instructions. It can decode 32-bit architecture as the following bytes sequence (BeaEngine, 2013)

| 0x89, 0x94, 0x88, 0x00, 0x20, 0x40, 0x00 (byte sequence in hexadecimal) |
|---|

It can print back on AT&T syntax

| Movd %edx, %ds:402000h(%eax,%ecx,4) ←(converted instruction in AT&T syntax) |
|---|

Moreover, the result on MASM32 syntax is

| Mov dword ptr ds:[eax + ebx*4 + 402000h], ed ←(converted instruction in MASM32 syntax) |
|---|

### 2.2.3. Capstone Engine

Capstone is a lightweight multi-platform, multi-architecture disassembly framework implemented in pure C language. It is an ultimate disassembly engine for binary analysis and reversing in the security community. It has many features like high performance, lightweight, simple API, details on disassembled instruction (decomposer). It is widely used in reverse engineering and disassembler applications. We are using it in our research as external Python library to convert the hex data to assembly. (Capstone, 2010)

## 2.3. Polymorphic NOOPs Engines

In this section we are describing the most polymorphic NOOPs Engines used by hackers that they bypass networks security and firewalls by converting the Shellcode to polymorphic that the security tools could not track that this data flow is a vector attack. We are using here using the engines that can reshape the NOOPs sled to unknown pattern.

### 2.3.1. ADMmutate

It is a tool created in early 2001 that allow the attackers to obfuscate any Buffer overflow vector attack the coder of this tool K2 and w00w00. The main purpose of this tool was to change the exploit signature every time it is executed which we know it results as "Polymorphic Shellcode." One of its technique is to change NO operation instruction to an equivalent instruction of 0x90 they always replace the NOOP section with 55 NOOP instruction possible; This way allow the attacker bypass IDS because the signature is changing each time. (SANS, 2002)

### 2.3.2. CLET

Convert the NOOP section to multi-bytes no operation and XOR encryption the payload body with using JUNK Bytes to defeat spectrum analysis of the data mining.

### 2.3.3. Metasploit

It is a computer security framework that executes vulnerabilities exploits against remote target machine and widely used in penetration testing world. It has many encoder's engines to encrypt the payloads also it provides many engines to encrypt and make the NOOP section polymorphic. It has two engines for an x86 processor that can convert the NOOP section to polymorphic SINGLE-BYTE and OPTY2. The first one is single-byte NOOP on this engine they got the ADMmutate 55 NOOP equivalent and added to them more of instructions to make the total of single byte 67 instruction. The second engine is OPTY2 that can create a multi-byte NOOP sleds with different length, and it has more efficient that CLET in this feature. So Metasploit NOOP generates a sequence of bytes of arbitrary length that equivalent to tradition NOOP sled (a sequence of 0x90 bytes) without having any predictable pattern

to bypass the IDS/IPS signature scanning of common NOOP Sleds. (Rapid7, 2013) (Burns, et al., 2007)

## 2.4. Machine Learning Tools and Libraries

### 2.4.1. Scikit-Learn

It is an open source library with simple and efficient tools in doing data mining and data analysis built for Python usage applications. It provides a range of supervised and unsupervised learning algorithms via the consistent interface in Python. This Library built upon SciPy (Scientific Python) that need include many libraries like Numpy and MAtplotlib, so we use Anaconda python version which has all of the required libraries. (Cournapeau, 2007) (Brownlee, 2014)

### 2.4.2. Anaconda

It is a leading freemium open data science distribution of Python for large-scale data processing, predictive analytics, and scientific computing, that aims to simplify package management and deployment. We used this distribution to reduce the python deployment complication.

## 2.5. Data Mining

Data mining is the process of different queries and getting the useful information that's not previously known or unexpected (Khan, Thuraisingham, & Masud, 2011). "It refers to the nontrivial extraction of implicit, previously unknown and potentially useful information from data in databases" (Zaïane, 1999);

While data mining and knowledge discovery in databases (KDD) are usually treated as substitutes, data mining is a part of the knowledge discovery process as shown in Figure (**2.7**) it consist of sequence of steps as following:

1. Data cleaning (remove noise).
2. Data integration (multiple data sources may be combined).
3. Data selection (data relevant to the analysis task are retrieved from the database).

4. Data transformation (data are transformed or consolidated into forms appropriate for mining by performing summary or aggregation operations).
5. Data mining (essential process where intelligent methods are applied to extract data patterns).
6. Pattern evaluation (identify the truly interesting patterns representing knowledge Based on some interesting measures).
7. Knowledge presentation (visualization and knowledge representation techniques are used to present the mined knowledge to the user) (Han & Kamber, 2005)

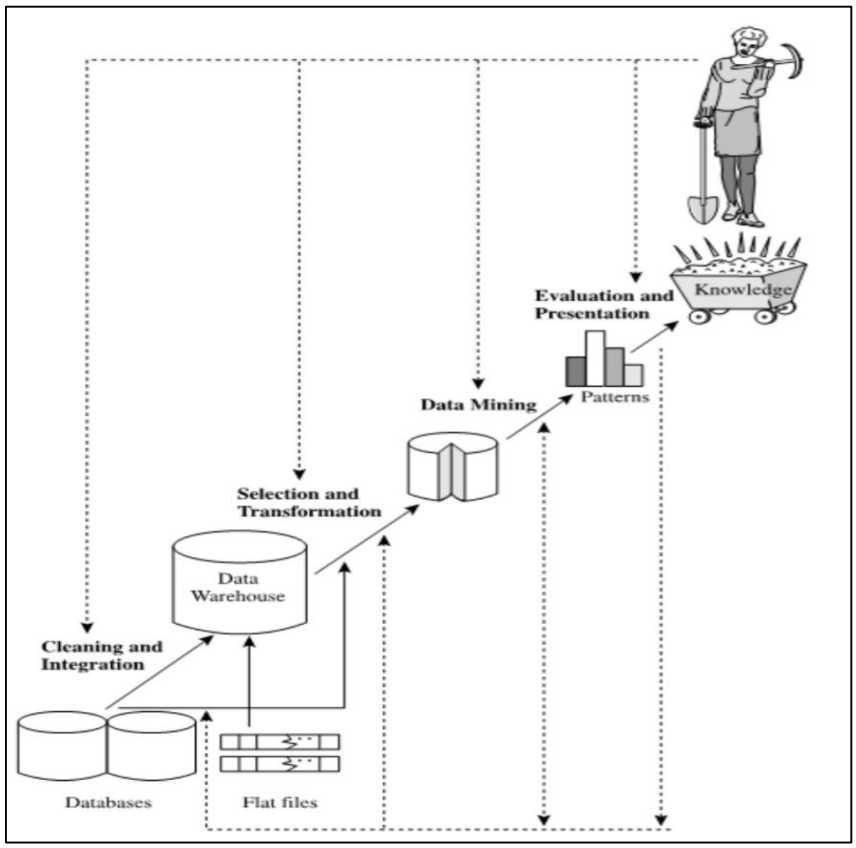



**Figure (2.7):** Data mining process steps (Han & Kamber, 2005)

Data mining studying areas include:

- Association – find patterns which something is connected to another.
- Sequence or path analysis - searching for patterns where something leads to following things.

- Classification - mining for new patterns and label all relevant objects with each other.
- Clustering - visually groups things that related which not previously identified.
- Forecasting - realizing patterns in data that can lead to reasonable predictions.

Data mining technology is used in many research areas, including mathematics, cybernetics, genetics, marketing, and security.

### 2.5.1. Supervised and unsupervised

Machine Learning is a type of algorithms that is data-driven, i.e. unlike "normal" algorithms it is the data which "tells" what the "right answer" is. A machine learning algorithm would not have such code definition, but would "learn-by-examples": you will show several malicious data, and the exemplary algorithm will eventually learn and be able to predict the class for the new data if it is malicious or clean.

This particular example of our situation is supervised, which means that examples must be *labeled*, or explicitly say which ones belong to our class and which ones are not.

In an unsupervised algorithm samples are not *labeled*, i.e. we do not say anything. In such a case the algorithm itself cannot "invent" what class it belong, but it can try to cluster the data into different related groups. (Vento, 2016)

The proposed solution intends to use supervised learning because in our case we have two labels of malicious data and clean data.

### 2.5.2. Data Mining Classifications methods:

In our research we evaluating a variety of classification methods against our feature extracted such as: Naïve Bayes (Bernolli & Multinomial), Support Vector Machine (SVM) and Decision Tree.

#### *2.5.2.1. Naïve Bayes:*

Naïve Bayes classifier based on Bayes' theorem, one of the main advantages of NBC is it doesn’t require large dataset of training set to find the means and variances of the

variables needed for classification. We used Multinomial and Bernorlli methods of this algorithm. (RapidMiner company, 2016)

#### *2.5.2.2. Support Vector Machine (SVM):*

Support Vector machine is supervised learning methods that analyze data and recognize patterns, it's used for classification and regression analysis (Eswari & Gunasundari, 2013).

#### *2.5.2.3. Decision Tree*

Decision tree is one of the supervised learning algorithms that follow the "Divide and conquer" approach to solve the problem by learning from autonomous cases (Ian & Witten, 2005).

The structure of tree includes: root node, branches and leaf, each node represent a test for an attribute, and the branch fork the result, and each leaf node represent a class label as shown in **Figure (2.8)** (Tutorials Point, 2016).

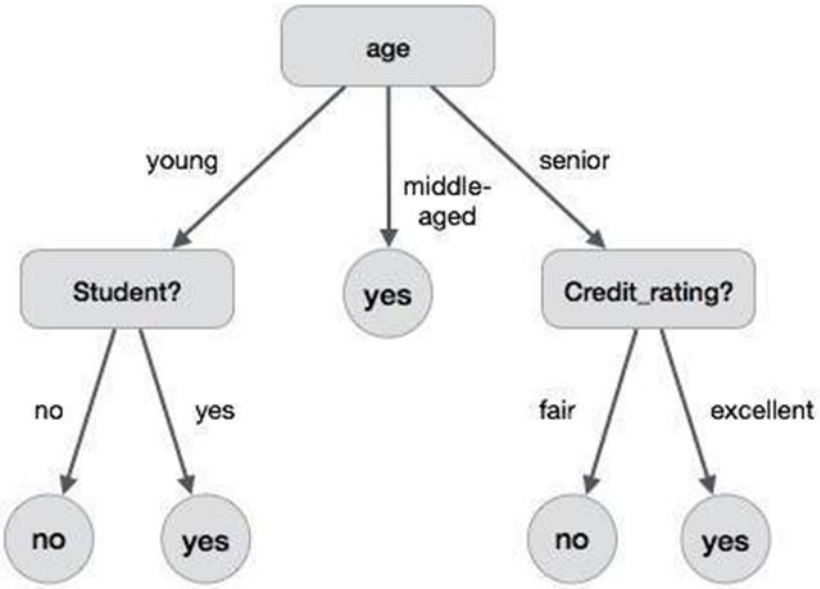



**Figure (2.8):** Decision Tree

#### *2.5.2.4. Stochastic Gradient Descent*

A very efficient approach to discriminative learning of linear classifiers under convex loss functions. SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language

processing. It is efficient and easy to implement but it is sensitive for the feature scaling (Cournapeau, 2007).

#### *2.5.2.5. Adaptive Boosting*

It is a machine learning meta-algorithm(estimator) that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases (Cournapeau, 2007).

### 2.1.1. Evaluation Methods

#### *2.1.1.1. Confusion matrix*

The confusion matrix is a very useful method for analyzing how well our classifier can define and detect the different classes, its structure as shown in Table (2.1).

**Table (2.1):** Confusion Matrix

| | | True Class | |
|---|---|---|---|
| | | +ve | -ve |
| Predicted Class | +ve | TP (True Positive) | FP(False Positve) |
| | -ve | FN (False Negative) | TN(True Negative) |

- True positive (TP) refer to positive instances that correctly labeled.
- True negatives (TN) refer to negative instances that correctly labeled.
- False Positive (FP) are the negative instances that were incorrectly labeled.
- False Negative (FN) are the positive instances that were incorrectly labeled.

#### *2.1.1.2. Performance Measures*

From the confusion matrix, we can estimate and calculate the accuracy, recall, precision, and F-measure which is used in evaluating the performance of the classification method. This performance metrics used in chapter five to evaluate the proposed solution.

## 2.2. Dynamic, Static, and Hybrid Analysis

Static analysis is performed without runtime execution. Static analysis tool inspects program code or in assembly for all possible run-time behaviors and seek out coding weakness, back doors, and potentially malicious code. Dynamic analysis acts the opposite approach and is executed while a program is in operation. Dynamic test monitor system memory, functional behavior, response time, and overall performance of the system. The hybrid analysis combines the two mentioned analysis. (DuPaul, 2013)

## 2.3. Summary

In this chapter, we have described what is the most techniques and tools used in our research. Beside that we discussed what is the buffer overflow and the Shellcode with why it's related in our research and where is the NOOPs sled located and why it is used. We have used as our primary development environment the Anaconda which supported all the libraries we need in writing the scripts like Python, NLTK and the valuable library Scikit-Learn, which we used it all the time in classification. We used Metasploit to generate the malicious dataset and select features using Capstone engine. Also, we mentioned about the data mining techniques which we are using in the proposed solution and which type we are using it (supervised). Finally, referred to about the confusion matrix and how we use it in evaluating the performance of the data mining algorithms.

# Chapter 3
# Related Work

Overflow detection and prevention problem have been studied since the mid-nineteenth. However, many modern types of researches have been published to solve this hot problem. We have review many researches and their approaches to deal with this attack. There was many types of analysis, we are illustrating in this chapter these types in sections like describing the research of some in the field of static analysis which analyze the Shellcode statically and predict if it is malicious or clean. Others used dynamic analysis which try to detect Shellcode using analyzing the packets in real execution environment. Also, quantitative analysis used by studying the polymorphic engines and how it works. Finally, illustrate the Hybrid way by using the static and dynamic in mixture to detect the Shellcode.

## 3.1. Static Analysis:

(Gamayunov, Quan, Shakharov, & Toroshchin, 2009) proposed Racewalk algorithm which is a significant modification of the Stride algorithm (Akritidis, Markatos, Polychronakis, & Anagnostakis, 2005) which had linear computational complexity, they claim novelty of NOOP-sled detection using IA-32 instruction frequency analysis and SVM-based classification, this approach reduces the false positive and the speed of operation is 1Gbps, main idea in this algorithm is there NOOP-zone which consists of generally useless instruction to allow the return address zone be in the correct stack segment because this varies from system to system, so they detect the sled candidates and sent them to SVM-based instruction frequency analyzer. Using only Four Shellcode engine generator they applied this algorithm. Still there many defects like detecting NOOPs of IA-64 and couldn't detect the Shellcode that construction methods do not rely on NOOP-sleds or used Self modified sleds not supported and bypassed by spoofing classifier in same instruction set but with unusual operands.

(Pasupulati, et al., 2004) have proposed "Buttercup" SNORT plug-in that can counter against polymorphic buffer overflow exploits by targeting 19 return address ranges that buffer overflow exploits, They assumed that the encrypted shellcode would change every other bit in the payload packet to avoid detection but there critical part in payload couldn't be encrypted this part is the "return" memory address. Simply they identify the ranges of the possible return memory addresses for existing buffer-

overflow. The return address ranges they were collected from various Buffer Overflow vulnerability that affected many operating systems. As their evaluation results in excellent detection for Shellcode, but there are many drawbacks in their solution because may not be detected if there is a miscalculation in the "range offset" and "range depth" values; beside that including the return address ranges may be changing according to operating systems updating or upgrading that will get with it more ranges that the system couldn't know about it and the attackers do.

(Akritidis, Markatos, Polychronakis, & Anagnostakis, 2005) have designed new sled (sequence) detection heuristic called STRIDE that detects several types of sleds that have significantly more computationally efficient which can be used in networks. So their demonstration depend on detection heuristics can be thwarted by more elaborate sled obfuscation techniques like NOOP instructions, One-byte NOOP-equivalents, Multi-byte NOOP, Four-byte Aligned, Trampoline-sled, obfuscated Trampoline-sled. By searching for every position of the data to find a sled. Despite STRIDE can detect several classes of sleds that cannot be identified by other solutions, the low false positive rates, but it still suffer from some weakness if the attacker does not use sled in the payload or use self-modifying sleds, and processing time very exhaustive beside that STRIDE could not detect the payload attack if there new equivalent NOOPS long bytes as they have restricted space of equivalents.

(Hsu, Guo, & Chiueh, 2006) present Nebula system which works as network-based buffer overflow attack detection that can detect both known and zero-day buffer overflow attacks based on packets analyzing without modifications on the hosts. By using the generalized signature to capture all known buffer overflow attacks to reduce the false positive to a negligible level. So the main signature that Nebula uses to detect buffer overflow attacks is a sequence of identical 4-byte words that correspond to an address in the stack region or text region, to reduce false positive rate Nebula recognize the FTP, HTTP, P2pfile sharing, and Bit Torrent and exclude bytes in downloaded files so this improves the optimization significantly. For overall design the proposed design for generalized signature is as following: if an input string contains a stack address that repeats N times, then it is regarded as code injection (CI) attack; if an input string contains at least N copies of a pattern that consists of a shared library function's entry point address followed by at least one stack address, then it is regarded

as return to Libc (RTL) attack and this algorithm depend on N =3. This design solves two types of payloads attack but couldn't handle ROP or non-ASLR attacks.

(Zhao & Ahn, 2013) Proposed a technique for modeling Shellcode detection and attribution through a novel feature extraction method called instruction sequence abstraction, which extracts coarse-grained features from an instruction sequence. This technique uses Markov model for Shellcode detection and support vector machines for encoded Shellcode attribution. There novel solution based on static analysis and supervised machine learning techniques, to extract coarse-grained features used instead of byte patterns, the instruction sequence abstraction. The evaluation shows that this solution can detect all types of un-encoded Shellcode from their dataset and can attribute encoded Shellcode to its origin engine with high accuracy. Despite the efforts that got our attention; but it has some weakness to IA-64 Shellcodes beside the small sample they used in training and all of this samples was from only one engine also using all Shellcode sections in the training because the model works on known payloads and range of it available for the researchers but it bypassed by adding low NOOP's all together with unknown payload in Shellcode so it can spoof it and pass.

(Wang, Wang, Luo, & Fang, 2007) Proposed DMPolD (Data Mining Polymorphism Detection) that can detect polymorphic exploit based on semantic signature and data-mining. The proposed method recognize JUMP address based on Bayes algorithm. The contribution was in building the mode of OSJUMP using online worms using specific JUMP addresses and based on this model analysis of features of polymorphic exploits and features of perfect ones, a then method to detect exploit through recognizing JUMP address using data mining. To prove there idea they implemented snort plugin (ODMSnort) and evaluated the approach on it, the results show DMPolD can detect polymorphic exploit with very low false-positive. Our opinion is supervised machine leaning to detect Shellcode depending on JUMP address on training could not detect all the worms or non-seen worms because the JUMP address may always be not using this JUMPs that they used.

(Masud, et al., 2008) Proposed DExtor a data mining based exploit code detector that protects network services. The they pivoting assumption that the normal traffic to network services contains only data whereas exploits contains code. Their system

trained with real data containing exploit code beside normal traffic after that put DExtor between a web service and its gateway firewall. Training consists of disassembly, feature extraction, and classification. The feature extraction depends on instructions count, instruction usage frequency, and code vs. data length. The data set used contained real exploit code as well as normal traffic to web servers. The evaluation applied on unencrypted exploits from Metasploit and encrypted using other nine engines to generate 1000 exploits and collect from internet 9000 exploit; this data set applied on different classifiers, and the results show very high accuracy and little false alarm rates. We see that according to the main assumption on DExtor which depends on main use for the network is transfer data and if there lots of instructions found on the network means it may include attack, but this is entirely untrue if we used the network in downloading binaries or executing some application from LAN nodes.

### 3.2. Dynamic Analysis:

(Polychronakis, Anagnostakis, & Markatos, 2006) present Polymorphic Shellcode detection method by emulate execution of every possible instruction sequence in Network Intrusion Detection System(NIDS) embedded CPU, aiming to identify the execution behavior of polymorphic Shellcodes, their approach relies on fully-blown Intel Architecture 32 bit(IA-32) CPU emulator. The execution of a Polymorphic Shellcode splits into the execution of two sequential parts: the decryptor and the actual payload. If an execution chain of an input stream during decoder decryption read the encrypted payload in order to decrypt it then, the system raises the alarm. As our review of this approach we found that this methodology only detects payloads that decrypt their body before executing their actual payload so the plain payloads couldn't be catch, also executing all the instructions will delay the throughput of the network, beside that the attacker can leverage context not available at the network level for building Shellcodes that cannot be unambiguously executed on the network level processor emulator. Detecting such attacks remains an open problem in this approach.

(Fen, Fuchao, Xiaobing, Xinchun, & Bing, 2012) present a method uses randomization based on data protection through protection of pointers and arrays, because of buffer overflow nature which depends on exceeding write on the limited area and populates the return address they use randomization on the arrays and pointers

in program space to protect buffers, point data, and return address. This randomization applied to the source by using XOR encryption for all the array and buffers, so when the overflow happened, the target will be an encryption value which couldn't point to, then the attack failed. This approach applied on the coding time to protect yourself application from using it in any type of buffer overflow attacks on the systems, but the major problem still available on the applications from the shelf or on the operating system itself.

(Khodaverdi & Farnaz, 2013) proposed robust run time heuristic for detecting those Shellcodes which hard-coded addresses as they take in consideration there still too many users using older versions of windows which not protected by Address Space Layout Randomization (ASLR) -enabled Windows. They used a custom emulator which supports the execution of IA-32 instructions, and they repeat the execution multiple times starting from each location of the input stream to find all possible executable sequences of instructions in the input stream and detect any hard corded address that points to the stack pointer. Their evaluation results show low false positive on 10 million random binary. They assume using this emulator in a host level to detect the attacks and for better performance. However, this approach could not detect return oriented programming (ROP).

### 3.3. Quantitative Analysis:

(Song, Locasto, Stavrou, Keromytis, & Stolfo, 2009) present a quantitative analysis of the strength and limitations of Shellcode polymorphism and describe the impact of these techniques in the context of learning-based IDS systems. They focused on two methods: Shellcode encryption-based and targeted blending attacks; because this two types used in the wild attack and successive in evading IDS sensors. Their paper demonstrates metrics to measure the effectiveness of modern polymorphic engines and provide insights into their designs. The paper dive in the construction of many Shellcode types to understand the overall issue, and after that analyzed the polymorphic engines –six of them- and by generating 10000 unique samples they plotted visualization images for each engine outputs to extract the pattern they use in creating the op codes, also they combines two engines that using polymorphism and blending in one engine called it A Hybrid Engine, they simply use CLET to

cipher the Shellcode, then hide CLET's decoder with ADMmutate and use ADMmutate's advanced NOOP sled generator and show how the attackers can blending between many engines to generate new patterns. After that presents new design to detect the modern obfuscation techniques. This paper allows us to go throw the internal of designing the polymorphic Shellcodes engines.

### 3.4. Hybrid Analysis:

(Yuan & Ding, 2011) Proposed a method that use's static analysis (source code analysis) with the dynamic test (test a program while it is running), so this approach strikes a proper balance between static and dynamic analysis to identify buffer overflow vulnerabilities in binary code (IA-32) without source code. They used two steps in their approach, first find some potential weakness locations then test every potential weakness locations so reduce the false positive. After disassembly programs they go throw many steps include identify function call relations, analysis stack space, analysis parameters, the use of local buffer, and finally determine the overflow function by using BugScam that can detect functions utilized in the binary file like Strcpy and so on and on the dynamic use Ollydbg to populate this functions that identified before in static to see if it check bounders or it overflow, testing results shows low false alarm. We see this approach can handle the stack overflow, and heap overflow can be a success and need from us to put all the binaries of the organizations to this analysis to allow it know if there is the ability to buffer overflow and this is not easy to be done.

### 3.5. Comparative Analysis:

(Silberman & Johnson, 2004) This paper examines two approaches by applying for a generic protection against buffer overflow attacks and critique the effectiveness of available buffer overflow protection mechanisms on the Linux and Microsoft Corp.'s Windows platforms. They explained the concepts behind buffer overflow protection software's and implementation details for popular systems, Discussed protections implementations in kernel enforced protection like MMU ACLs, NOEXEC, ASLR and protection in compiler enforced protection like Stack Canaries. After that describe how Linux and Windows use mixed techniques to protect from Buffer overflow. Finally shows attack vector test results for each technique that evade

buffer overflow according to the long list of different attack techniques. They find a final result that the currently available solutions may not be perfect to defense buffer overflow attacks.

## 3.6. Previous Solutions and weakness:

In Table (3.1) listed the related works solutions with weakness they are suffering.

**Table (3.1):** Related works and its weakness

| # | Solution Proposed | weakness |
|---|---|---|
| **1** | Use return address range from public worms to search for sleds have them to catch buffer overflow action. (Pasupulati, et al., 2004) | Range offset of used returns may change at any time. Bypassed by unseen return addresses. |
| **2** | Use static analysis to identify the 1-byte equivalent, multi-byte NOOPs by using n-gram disassembly. (Akritidis, Markatos, Polychronakis, & Anagnostakis, 2005) | Self-modified sleds not supported and processing time very exhaustive. Bypassed by new equivalent NOOPS long bytes as they have restricted space of equivalents. |
| **3** | Use instruction frequency analyzer to detect NOOP sled using classification algorithm. (Gamayunov, Quan, Shakharov, & Toroshchin, 2009) | Self-modified sleds not supported, no detecting for IA-64 and shells does not use NOOP sled, bypassed by spoofing classifier in same instruction set but with unusual operands. |
| **4** | Detect the packets if there any 4 bytes that represent stack address and repeated N times. (Hsu, Guo, & Chiueh, 2006) | Couldn't catch ROP or anti ASLR shells. Totally outdated in new versions of OSs. |

| # | Solution Proposed | weakness |
|---|---|---|
| **5** | Use instruction sequence abstraction for all the shells in Markov model for detect shells. (Zhao & Ahn, 2013) | The small sample used in training only one engine, beside bypassed by adding new unknown payloads in the Shellcode, so the model spoofed. |
| **6** | Assuming the packets in the network is data and when there code then its exploit and used data mining to classify. (Masud, et al., 2008) | Work only on web services. |
| **7** | Use data mining to recognize jump address based on Bayes by building a model using worms specific jump addresses and added the method as a plugin to snort. (Wang, Wang, Luo, & Fang, 2007) | Couldn't catch unknown worms nor all worms because jump addresses could change in anytime. |
| **8** | Use embedded CPU to execute the behavior of polymorphic Shellcodes. (Polychronakis, Anagnostakis, & Markatos, 2006) | Don't detect shells that decrypt body before execution, too delay in packets & IA-32 only. |
| **9** | Use randomization in the buffer by using XOR encryption for all data stored in memory so couldn't execute shells. (Fen, Fuchao, Xiaobing, Xinchun, & Bing, 2012) | Need to apply in coding time and couldn't apply to applications you do not have the source code. |
| **10** | Using return address to identify shells by emulate executable sequences. (Polychronakis, Anagnostakis, & Markatos, 2006) | Couldn't detect ROP types. |
| **11** | find a potential weakness in code and then test against BOF and after that use BugScam to identify all the vulnerable | Handle the stack overflow and heap overflow can success in case putting all the binaries of |

| # | Solution Proposed | weakness |
|---|---|---|
| | functions and use Ollydbg to populate this function and know if it is overflowed or not. (Yuan & Ding, 2011) | the organizations to this analysis to allow it know if there is the ability to buffer overflow and this is not easy to be done. |

## 3.7. Summary

As listed in this chapter. The researchers of detecting buffer overflow were using many analysis techniques. Firstly, Static analysis in using Return address range to catch buffer overflow. Beside identify the 1-byte equivalent, and Multi-byte NOOPs by using n-gram disassembly. Also, use instruction frequency analyzer to detect NOOP sled using classification algorithm. Identify the packets and if there any four bytes that represent stack address and repeated N times. Use instruction sequence abstraction for all the shells in Markov model for detect shells.

Secondly, Dynamic analysis assuming the packets in the network is data and when their code then it is exploited and used data mining to classify and use data mining to recognize jump address based on Bayes by building the model using worms specific jump addresses and added the method as a plugin to snort. On the other hand, there was a dynamic analysis that use embedded CPU to execute the behavior of polymorphic Shellcodes, use randomization in the buffer by using XOR encryption for all data stored in memory so couldn't run shells, and using return address to identify shells by emulate executable sequences. All of these researches have defects that we discussed on each listed approach, based on that our proposed solution will solve this holes in the scope we identified to detect the unknown polymorphic Shellcodes by using classification algorithm on the op-code of polymorphic NOOP sled.

As we mentioned there many proposed solutions that depend on data mining techniques but the difference between my work and all of these solutions that we using the NOOPs section in Shellcode and the type of data that extracted in feature extraction because we are using the operation code of the instructions for features.

# Chapter 4
# Proposed Solution and Methodology

In this chapter, we present and illustrate our solution in detecting the polymorphic Shellcodes using essential part NOOP's sled. This chapter is organized by defining our solution methodology steps followed by identifying our dataset and how we extracted and collected it from the Polymorphic Shellcode engines then how we preprocess and feature selection the data, then describe how we applied the classification algorithms, and finally applying used classifiers and evaluate the method. We have used as our primary development environment the Anaconda which supported all the libraries we need in writing the scripts like NLTK and the valuable library Scikit-Learn, which we used it all the time in classification. We used Metasploit to generate the malicious dataset and select features using Capstone engine.

## 4.1. Solution Steps

Our solution depends on using data mining classification techniques to define packets of data if they are malicious (polymorphic NOOPs) or not. We extracted special features which depends on the operation code of the assembly instruction of network data. So the steps shown in Figure (4.1) is the base steps of the solution methodology.

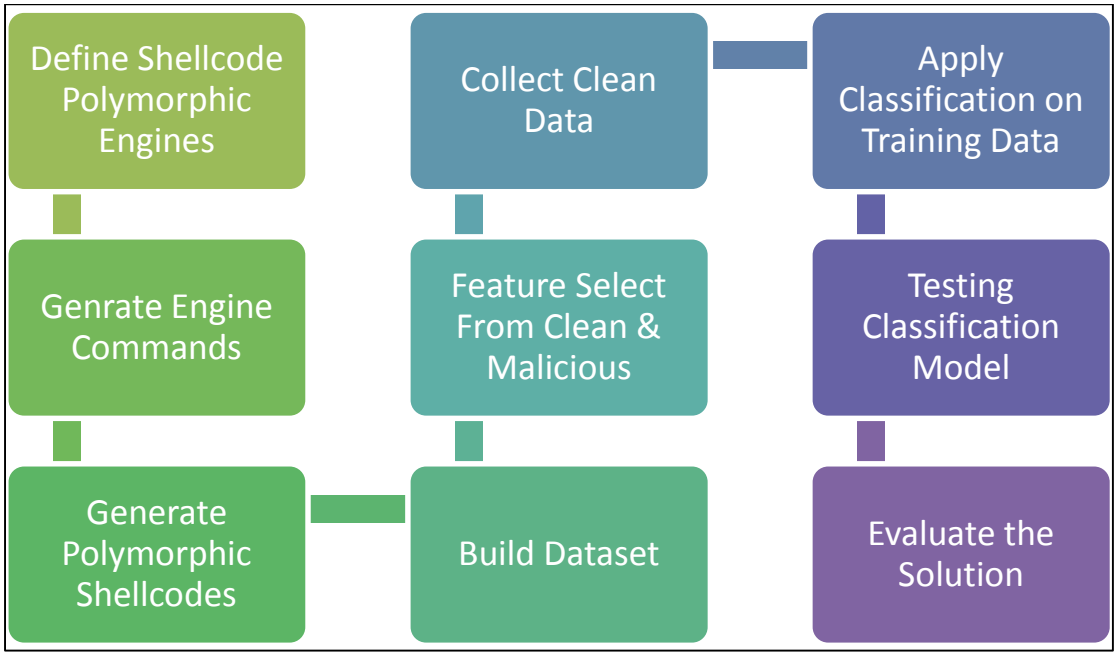



**Figure (4.1):** Solution Steps

In Figure (4.2) show the overall view of the solution to detect the malicious packets (polymorphic Shellcodes). We defined the polymorphic engines and collect the most popular Shellcode engines of Metasploit which its architecture is IA-32 like SINGLE-BYTE and OPTY2 engines. Then implement script that applies automatic generation on the engines with all possible parameters. So we generated a significant amount (500,000) of polymorphic Shellcodes so we can label this files as malicious because this engines is well known in the hacker's world that can generate CPU instruction that do nothing but not in the usual way we are writing in the assembly instructions. After that use Capstone Engine to disassembly all the NOOP sections. The last step in building the dataset is to extract the features that we will use in the classification algorithms, so we got the operation code of the assembly instructions. Also, repeat the process for clean data and applications files to build the equivalent dataset which labeled with clean.

The last step in the solution is to pass this two labeled dataset to the classification algorithm. We used classification methods such as SVM, Decision Tree, Bernoulli NB, Multinomial NB, AdaBoost, and SGD. Dataset separated to training dataset with 70% of the original dataset with balancing of clean and Shellcode data that we have we stopped using cross validation because of the large of the data and it take long time in training while substituting the features.

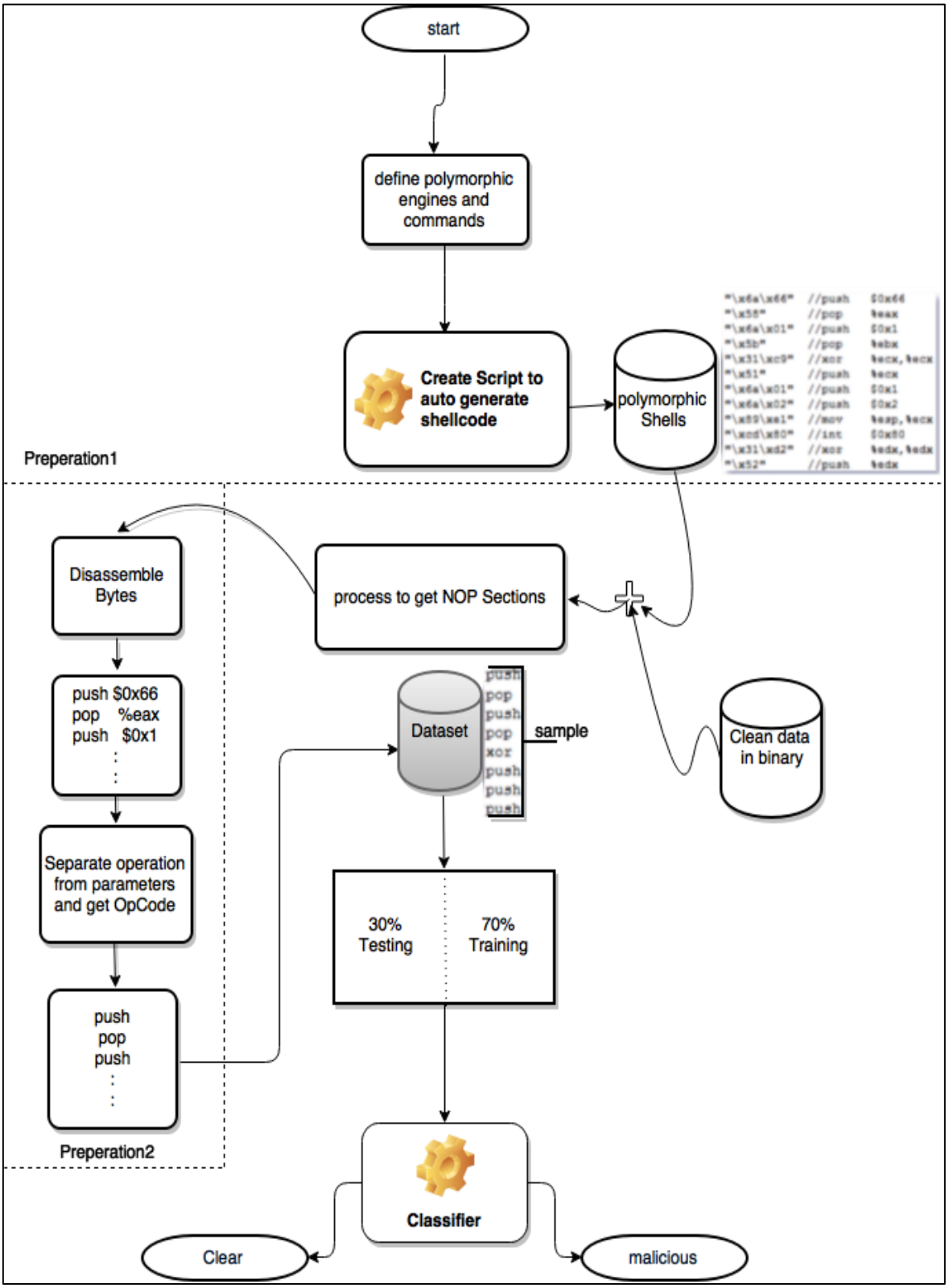



**Figure (4.2):** The Proposed Solution

## 4.2. Dataset

We start searching for online benchmark dataset of buffer overflow shell codes. Unfortunately, we could not find anything for this start, and we began emailing the researchers who have published papers in the area of our research, and only one professor responded with a negative response and told us that he does not have the dataset, but we can regenerate it from the exploits engines. Then we changed our direction to generate the dataset corpus ourselves, so we searched for all the engines which can generate polymorphic no operation instructions. We found that their many engines like CLET, ADMmutate, Metasploit-Single, and Metasploit-Opty2 (Zhao & Ahn, 2013).

After more investigating and searching we discovered that ADMmutate generates different 55 1-byte no operation, CLET can generate multi-bytes no operation. We concluded that Metasploit had all the 55 NOOPS of ADMmutate and added on it 12 new single bytes no operation so we excluding ADMmutate engine and we also found that CLET multi-byte included in the opty2 and excluded also.

Metasploit (single, opty2) engines used in generating the dataset, the steps employed in this stage as following:

### 4.2.1. Generate Polymorphic NOOP Dataset

We created automation script illustrated in flowchart Figure (4.3) to create Metasploit commands which can generate the no operation combinations of opty2 and single byte. This code creates a file (generator.rc) that hold all the commands that generate NOOP's from 1 byte to 5000 bytes and on each length it generates 100 different combinations of NOOP's so we have 5000* 100 = 500,000 command line like the sample shown in Table (4.1).

**Table (4.1):** MSF Command Sample with Description

| | |
|---|---|
| Commands samples generated | Use nop/x86/opty2; spool Desktop/nops/1000.nop; generate 1000 –t hex;<br>Use nop/x86/single_byte; spool Desktop/nops/33.nop; generate 33 –t hex; |

| | | |
|---|---|---|
| Describe the commands | Use nop/x86/opty2 | Use opty2 engine from Metasploit |
| | Use nop/x86/single_byte | Use single byte engine from Metasploit |
| | spool Desktop/nops/1000.nop | Set the file that stores the NOOP's |
| | generate 1000 –t hex | Execute command to generate 1000 byte encoded to hexadecimal. |

We executed this resource script (commands file generator.rc) on Kali Linux which has Metasploit installed by executing this terminal command

```
msfconsole –r generator.rc
```

After this command starts executed, this took to finish 160 hours and finally we have 500,000 files of polymorphic NOOP's in different sizes from 1 byte to 5 KB and 2.6 GB of the whole size. This is sample record of NOOP's in hex with length 30 bytes.

```
4e6bd547b22cbe1d15bbbf9325990c982d3f48b64090a8f8b5371c140441
```

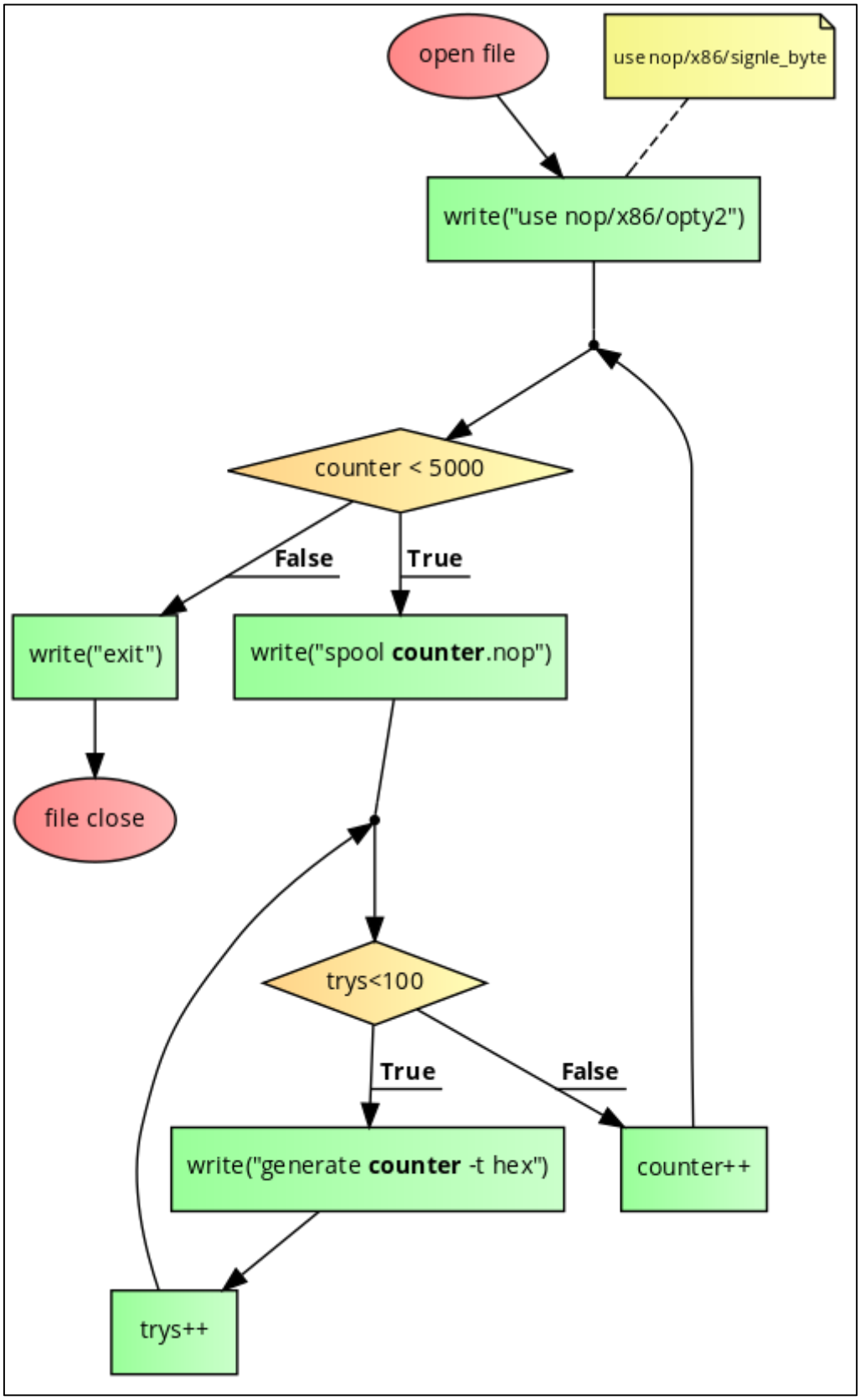



**Figure (4.3):** Flow Chart of the script that generates polymorphic engine commands.

### 4.2.2. Select Features from malicious Corpus

To select features from the polymorphic NOOP's generated we need to disassemble all the bytes that generated from the malicious Shellcode engines and get the full instructions which represent all these bytes. We used Capstone Engine to disassemble hexadecimal files to assembly then get the operation code of each instruction, so we wrote a script to do this conversion as shown in the flow chart in Figure (4.4).

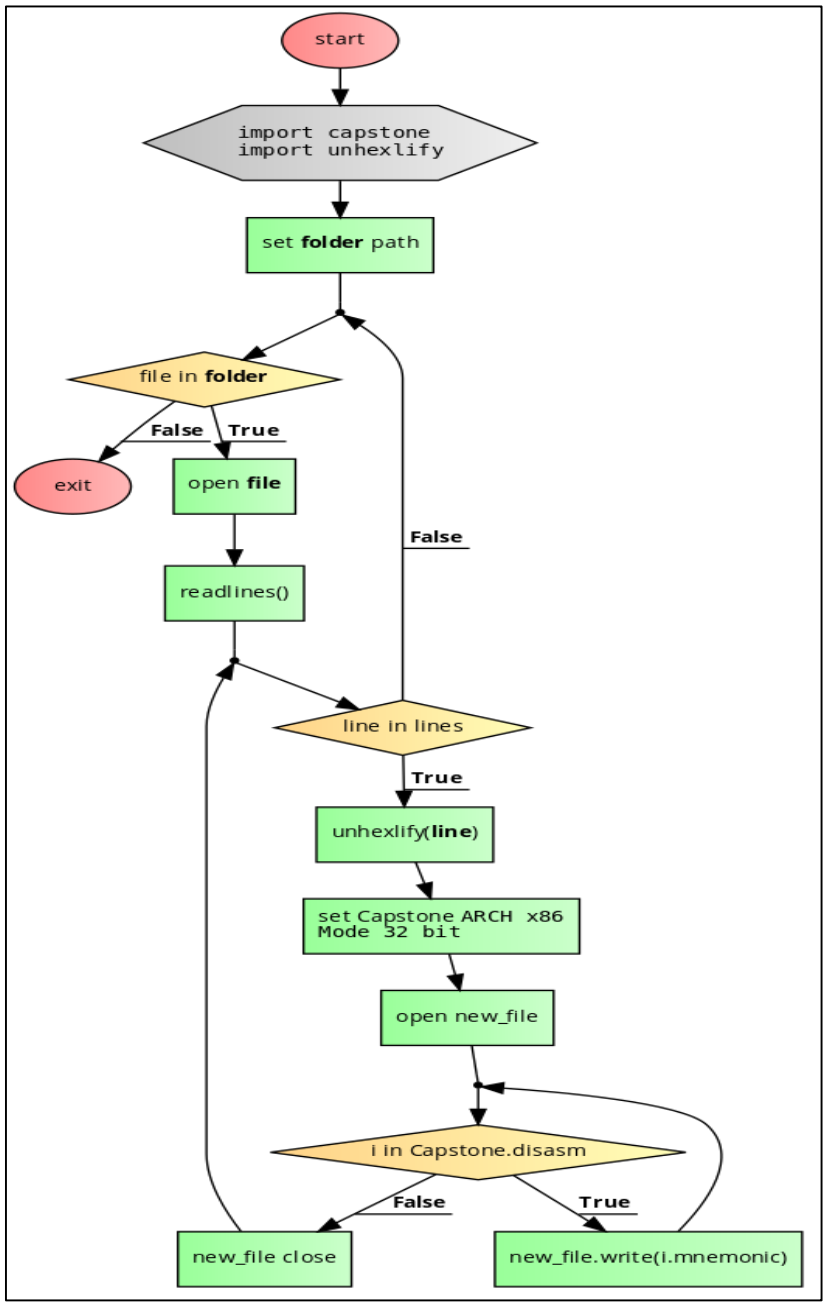



**Figure (4.4):** Feature Selection from Corpus

The procedure in Figure (4.4) read each hexadecimal files then convert to assembly instructions of Intel 32 bit architecture. Then on the next steps we extract the operation code only from the instruction line. This procedure take 3 hours to finish in execution to convert 500,000 assembly file, so we get finally the NOOP's dataset with size of 4.2 GB.

### 4.2.3. Create Clean Corpus

In order to do supervised classification, we need to create the second label to let the method work. Here we need create clean data corpus with "clean" label for classifier methods. We have been collected 120,000 files from different types such as Movie clips, books, images, texts, docs, applications, binary libraries, Dll files, Etc. The final size of these files was 3.7 GB. Then next step is to convert this clean files to assembly and then select the features as we did with the malicious corpus, so we created a new script which can convert binary data to assembly using Capstone Engine as shown in flowchart Figure (4.5).

As description of the Figure (4.5) we have collected all the clean data paths and read them in binary to provide them as input to Capstone Engine which processing this binary data and convert them to assembly instructions then select the main features that we need which is operation codes like what we did with NOOP's data files before.

According to the most of the clean raw files -which was input to the system- does not a CPU instruction so the Capstone Engine could not found appropriate instructions on the IA-32 that can be correlated with the bytes.

Finally, at this stage, we have a corpus of clean assembly operation code which has size of 737 MB for 72,000 files.

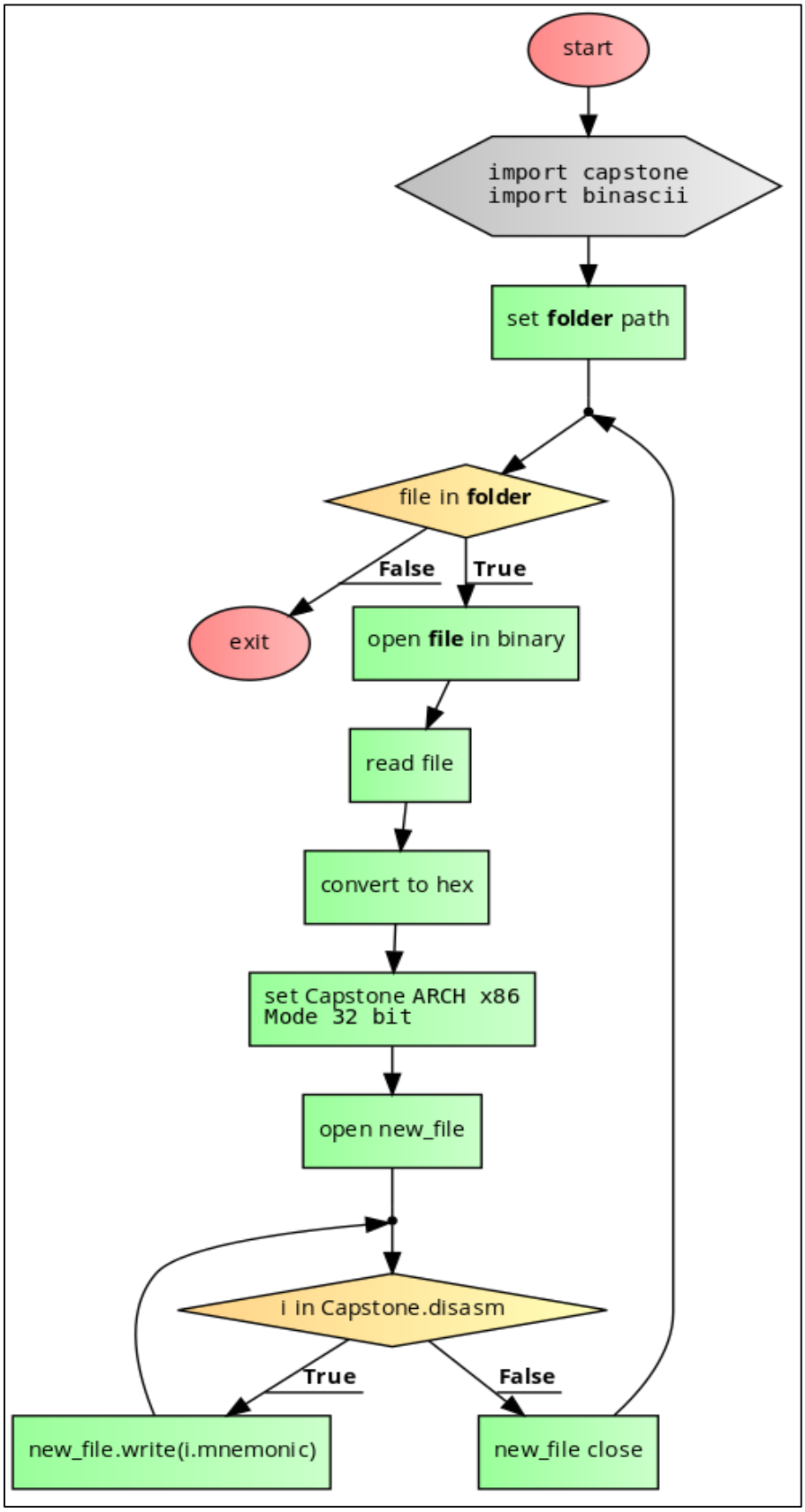



**Figure (4.5):** Generate clean corpus from clean files

### 4.2.4. Dataset sample

In this section shows table which describe sample of the dataset and the dataset representation (weighing) model and show how the decision tree will represent this model to label the records to clean or malicious. Datasets used is length independent and location independent on the byte sequence because of disassemble most shellcode bytes in the packets, no matter where it is started, and the solution approach to moniter data stream online so the length of packets or the location of the feature couldn't be known. The input matrix in the classification algorithm built depending on the next formula which represent each node of the matrix in binary.

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

Where,

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

$j$ is the index of feature records.

To give an example how is the formula working and how features input the model and what is the output represintation in an example. In Table (4.2) listing four samples of what is inputing to the system with labeling on each of this sample records that will be converted to boolean weighing matrix.

**Table (4.2):** Four Samples of Dataset

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

Next step is collecting all the features (instructions) occurs in the samples and get each feature only once (as we do not care to the order of the feature according to the reason mentioned in this section) without repeating from all of the samples that act as input to the system so it will appear as output of the above four samples as shown in Table (4.3) this features indexed according to the position where it is placed to act as matrix header.

**Table (4.3):** Feature names header

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

In this step after list all the features, need to transfer features to Boolean matrix by checking each feature in the feature names header is available or not, and if it is available in the sample it will be replaced with 1 and if not with 0 according to the formula.

**Table (4.4):** Matrix of Boolean Weighing of Four Example Records

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

In Table (4.4) binary matrix represented which act as the input to the classification algorithm that result by applying the previous formula with respect that the first two samples are clean label and the other malicious label. By using this sample matrix as input matrix for Decision tree algorithm to build the decision tree model that allow seeing the output visually. The result of this four records can be seen in Figure (4.6) which it classify two of them to malicious and the other two to clean. The tree shown in Figure (4.6) produced from the matrix in Table (4.4) using partitioning the examples recursively by choosing one attribute each time to find the best attribute installed in the root, then split data and find the best attribute in each node, then repeat this stage until all node are pure and the nodes contains fewer cases. By applying this building tree strategy on the matrix in Table (4.4) found that *aad* not fitting the best node at root. So, continue to the next *adc* to find it can classify all the samples from the next nodes as the first and the second samples have *1* and the other two have *0* this mean any sample that contains *adc* is a clean sample and malicious otherwise. This can be represented by classification rules like *(If adc <=.5 then label=malicious otherwise label =clean).* In conclusion of that the algorithm stops because it classify all the records with minimal nodes. Explaining about attribute Gini in the tree is a measure of how often the chosen element would be incorrectly labeled in each node. So, it reaches its minimum (zero) when all cases in the node fall into a single target label. The nods are not too many in the proposed example because of the samples is too little. The nods increasing as well as the records are increasing so the output model of 28

samples as another example as shown in Figure (4.7) have more nodes and so on in large dataset.

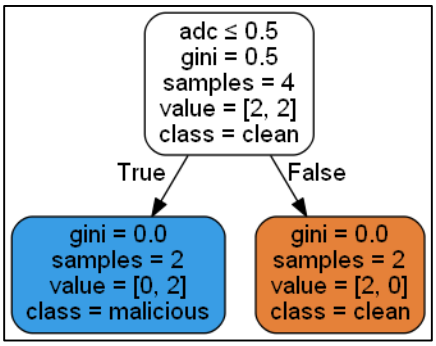



**Figure (4.6):** Output Representation of Decision Tree Applying on the four Samples

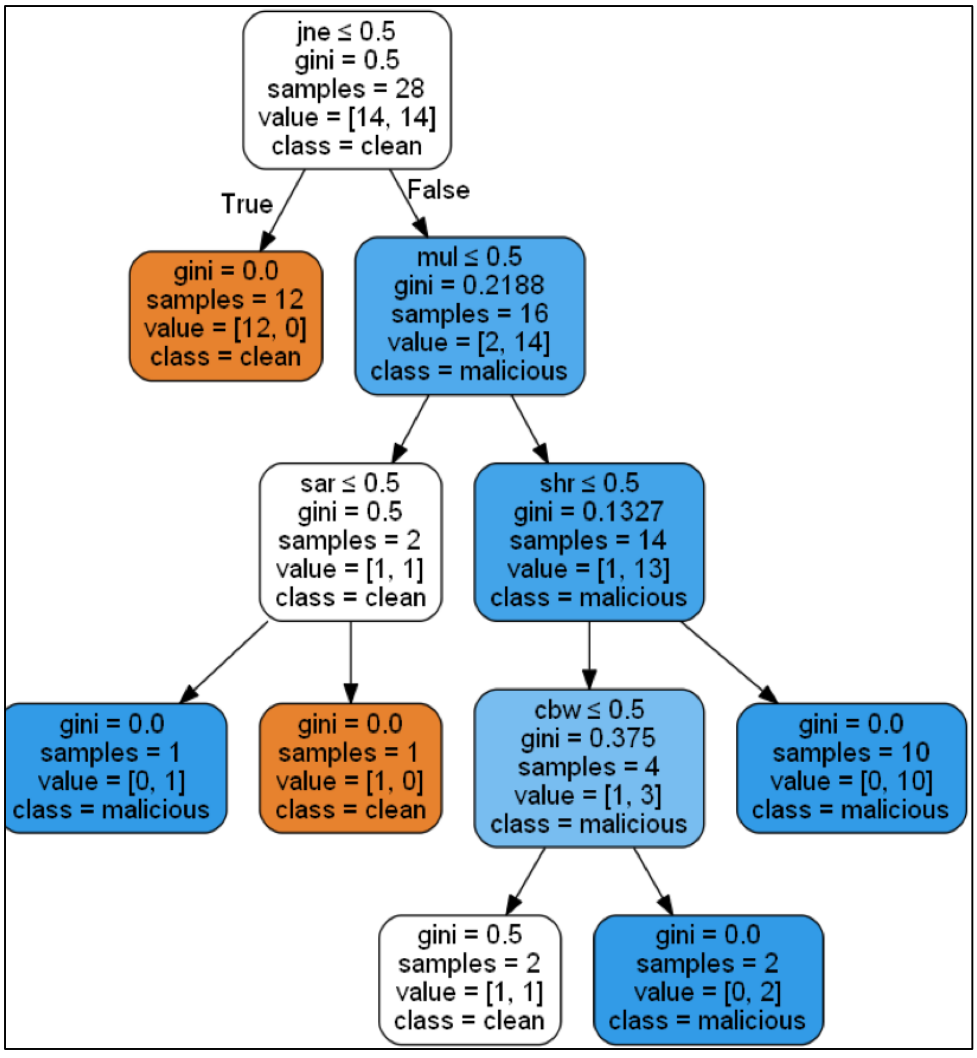



**Figure (4.7):** Decision Tree Model for Twenty eight Samples

## 4.3. Classification Procedures

To do supervised classification we need training data for the different classification algorithms to produce the model that can identify the related patterns between each class label then, we need testing data that we will apply the model to evaluate how is this model is correct by using different metrics like accuracy, recall, precision, and F-measure.

We have prepared in the dataset section the two labeled (clean, malicious) datasets to add them as input to the classification process. We have separated each data set from the two label's to 70% of data as training data for each and 30 % of data for testing the model. The 70% of malicious data is larger than the clean, so to achieve equality between clean and malicious dataset's we have shuffled all the malicious dataset randomly and separate 70,000 records from the original, and we have second reason to get 30% of the malicious corpus is to reduce the time of processing in different classification stages.

### 4.3.1. Preprocessing

On the first part of doing the classification is to refining and polishing the corpus besides doing shuffling, getting a small piece of the dataset, labeling the sets and combine the 70% of training data together from clean and malicious and finally combine the testing sets with each other. Figure (4.8) show a flowchart for the preparation for classification. As show we are importing libraries and classifiers, then load the corpus data using lazy loader because of significant data, also configure the lazy loader to get the dataset files from two folders named as "clean" and "malicious." After that we continue to shuffle the corpus and getting 70,000 from the malicious corpus, then labeling each data record with its appropriate label. In addition to all of that, we get 70% of each of the two datasets and combine them like training data and get the 30% as testing data. The last step is to pass this data to a procedure to let any Classifier algorithm to process this data.

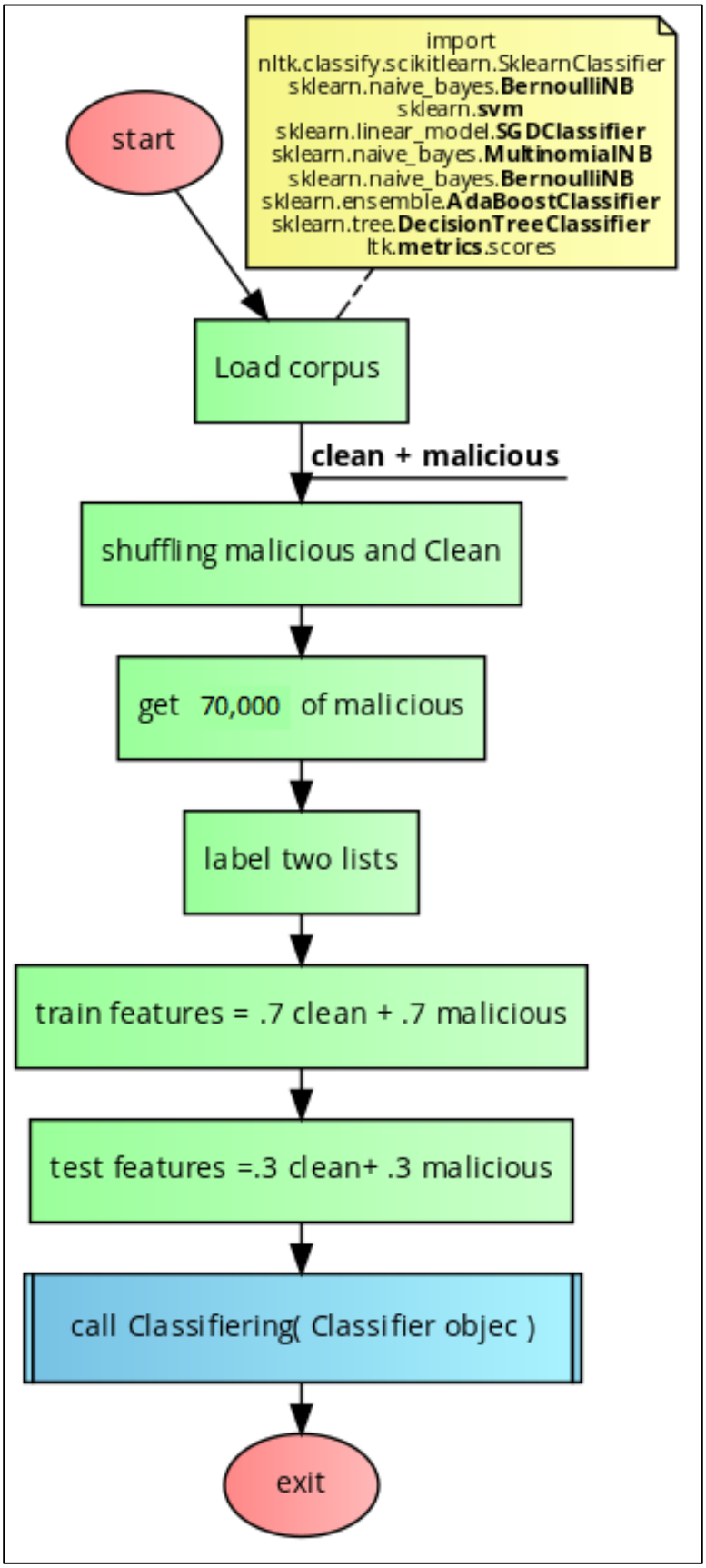



**Figure (4.8):** Corpus preparation

### 4.3.2. Classification process implementation

We have created a procedure which accepts as input any classifier algorithm implementation class which SKLEARN provided on its library package. In this stage we continue processing the data after we finished from preprocessing it, so we have a dataset for training ready and dataset for testing ready. Figure (4.9) shows our function in flowchart which can describe the flowchart as the following steps:

- The function has parameters to accept the Classifier method, training dataset, and testing dataset.
- Initialize the Classifier method with the different parameter variables which the algorithm deals with.
- Train the classifier method to get the model that we need to test.
- Compute the execution time of the training data processing.
- Test the trained model with the testing dataset.
- Compute the execution time of model test process.
- Compute the accuracy of testing model process results.
- Create reference set with originally labeled set and compute each feature of the testing data label using the trained model; to pass these two sets in the different evaluation metrics.
- Compute evaluation metrics (Precision, Recall, and F-measure) for the clean and malicious tested on the model.

Separating the assessment for each label from a security perspective to know how is the solution is efficient in detecting the attacks and know the rate of false alarm which is clean.

We used the described flowchart in Figure (4.9) to evaluate many classification algorithms to find the best method that support our solution from security point of view, these algorithms which we compared between were Support Vector Classification (SVM), Stochastic Gradient Descent (SGD), Multinomial Naïve Bayes, Bernoulli Naïve Bayes, AdaBoost classifier, and Decision Tree (DTs).

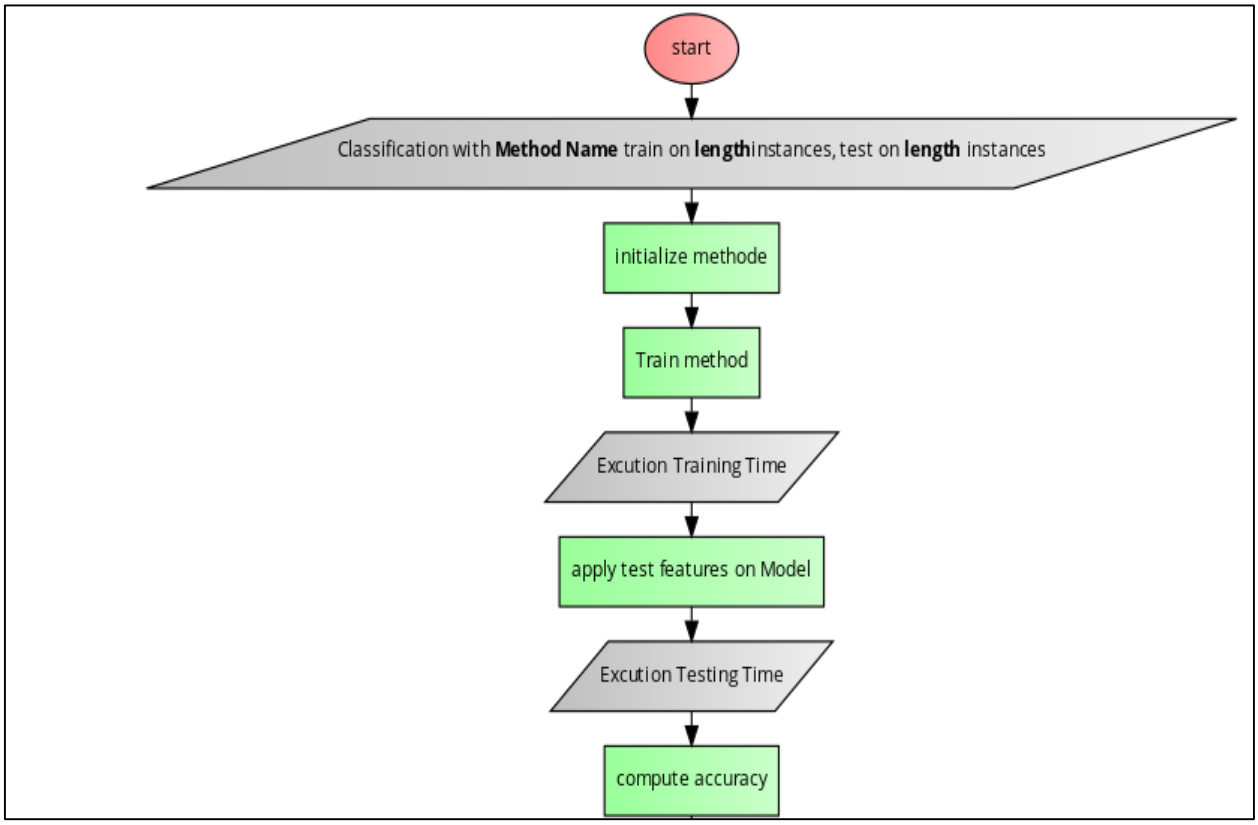



**Figure (4.9):** Classification training and testing processes and compute evaluation metrics

## 4.4. Summary

We have searched for a benchmark dataset to apply the proposed solution with; but we had not succeeded, so we have generated the dataset of the clean and malicious by ourselves and used the op-code of the NOOP's as feature selection. Next, we developed a script that allowed us to use the classification algorithms which implemented in the Scikit Learn library. We utilized in the script the shuffling then do generate the model of the classifier from training dataset then test this model, after that we compute all the performance metrics which we got it from the confusion matrix.

# Chapter 5
# Experimental Results and Evaluation

In this chapter, we are listing experimental environment, all the tools we have used to finish the work in this research besides explaining difficulties that faced us in the research, and classification settings we used.

Then viewed all the experimental results that we have performed on five classification algorithms. Finally, compare evaluation metrics that represented from the confusion matrix to choose the appropriate algorithm which can fit the best of security solution to apply it to our solution.

## 5.1. Experimental Environment

We have used Virtual Windows 7 64-bit, a processor with 2.5GHz quad-core Intel Core i7, RAM memory 10036MB, with 57GB of SSD hard Drive.

## 5.2. Experiments and Results

### 5.2.1. Experiments

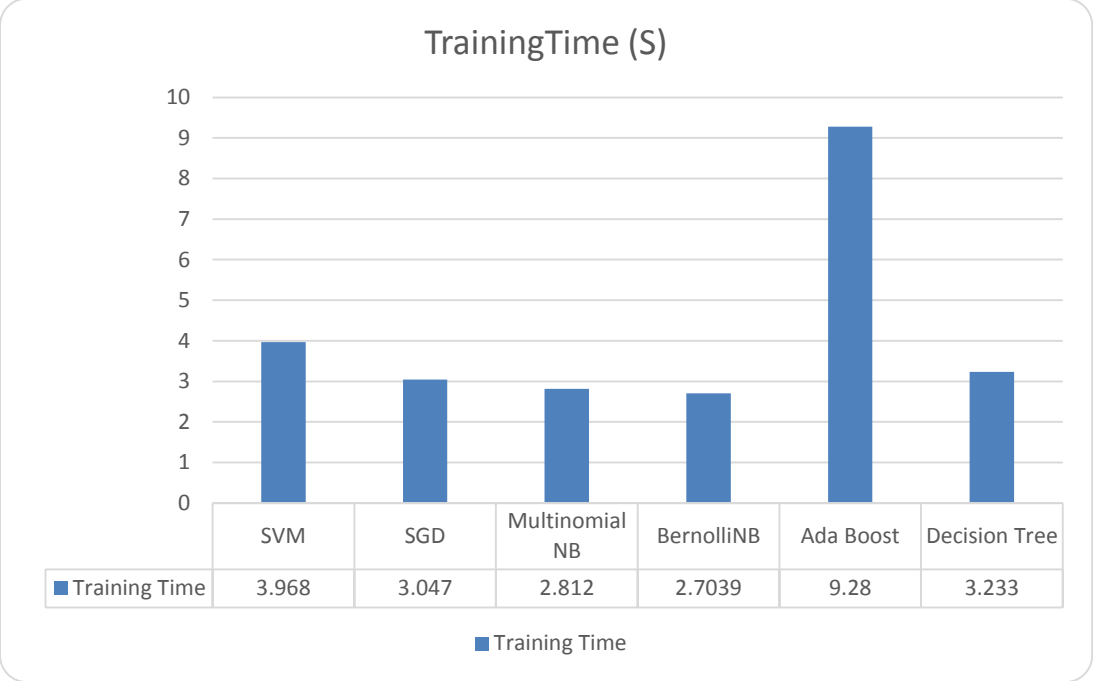
We started the experiments on the previously mentioned environment and classification settings. Table (5.1) illustrate the results of all experiments done with SVM, SGD, Multinomial NB, Bernoulli NB, Ada Boost, and Decision Tree and shows the results of different metrics like accuracy, recall, precision, and f-measure.

**Table (5.1):** Performance results

| | | SVM | SGD Classifier | Multinomial Naïve Bayes | Bernoulli Naïve Bayes | Ensemble Ada Boost | Decision Tree |
|---|---|---|---|---|---|---|---|
| **Execution Time (sec)** | **Training** | 3.968 | 3.047 | 2.812 | 2.7039 | 9.280 | 3.233 |
| | **Testing** | 1.734 | 1.389 | 1.391 | 1.6710 | 1.578 | 1.405 |
| **Accuracy** | | .94916 | .9399 | .9433 | .9366 | .9408 | .9333 |
| **Precision** | **Malicious** | .91271 | .9119 | .9018 | .9093 | .9218 | .9193 |
| | **Clean** | .99268 | .9645 | .9944 | .9678 | .9616 | .9482 |
| **Recall** | **Malicious** | .99333 | .9666 | .99126 | .97 | .9633 | .95 |
| | **Clean** | .905 | .9066 | .8916 | .9033 | .9183 | .9166 |
| **F-Measure** | **Malicious** | .95131 | .9385 | .9461 | .9387 | .9421 | .9344 |
| | **Clean** | .94681 | .9347 | .9402 | .9344 | .9394 | .9322 |

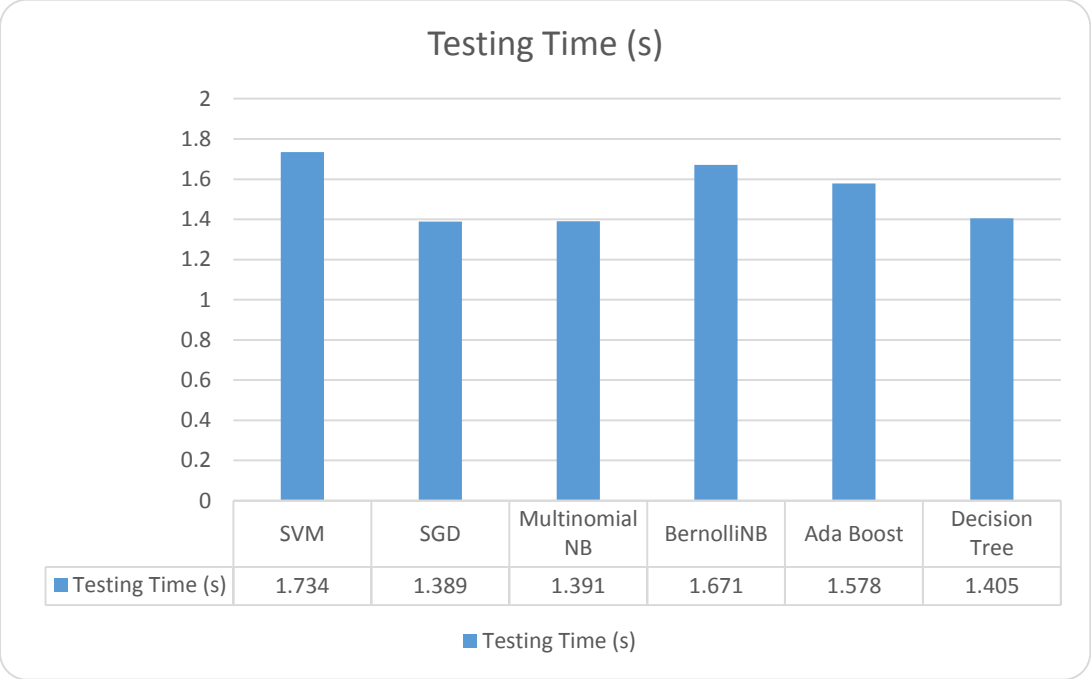
### 5.2.2. Comparing Results

**Table (5.2):** Experiments training time for all classification algorithms



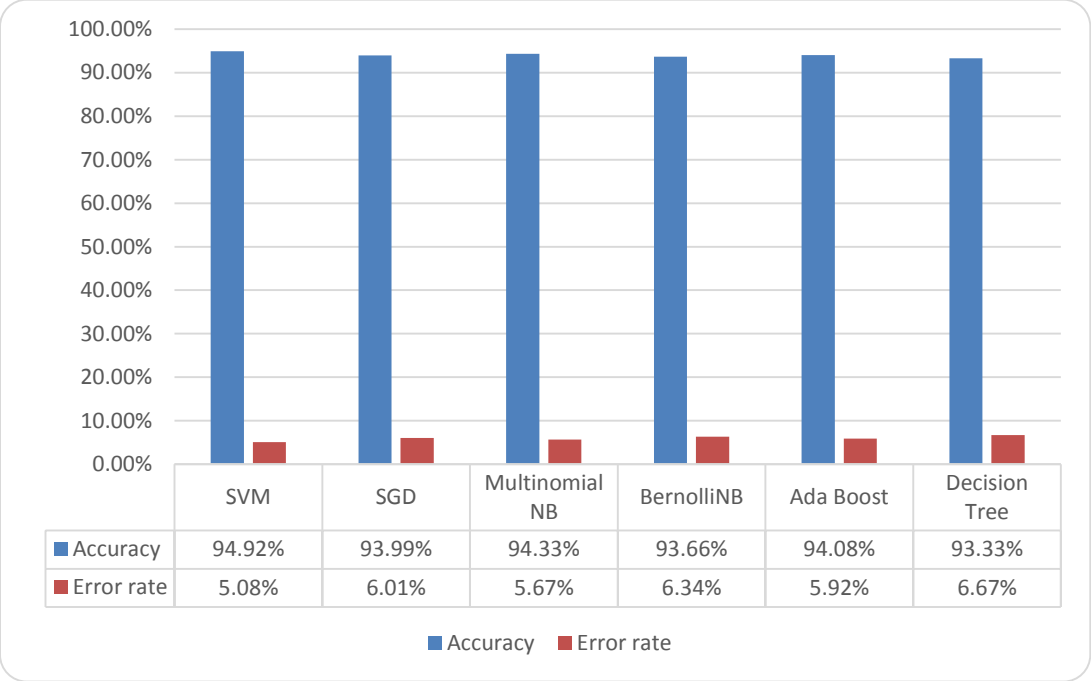

Execution time results of building models from the algorithms processes listed in Table (5.2). From this chart we can deduct that Bernoulli Naïve Bayes and Multinomial Naive Bayes are the quickest methods in contrast of AdaBoost is the slowest in creating the learning model by training the different methods. By the way, training time does not benefit I real use, so we could not depend on this metric as this would not be useful for us from security perspective nor let us choose the best here to get more successful results.

**Table (5.3):** Experiments testing time for all algorithm models




Execution time results for testing models generated from the different classification algorithms listed in Table (5.3) From the network security point of view we consider not affecting the data rate flow in the network when applying the classification model for instances at real time; So testing time is crucial as the detection system of polymorphic NOOP's will be on network flow so need not make this process take a long time when applying the classification on single instances. Founded that the results for all methods acceptable which we have at most 1.734 seconds to identify more than 150 MB of network data as malicious or clean. This indicates for the high speed result when applying the classification model on single packet instead mass of data in real environment. We have Multinomial Naïve Bayes, and SGD is the fastest and SVM is the slowest. This metric give us a view about the speed when classifying mass of data and that indicate that classifying single instance will be very fast.

**Table (5.4):** Accuracy of experiments results



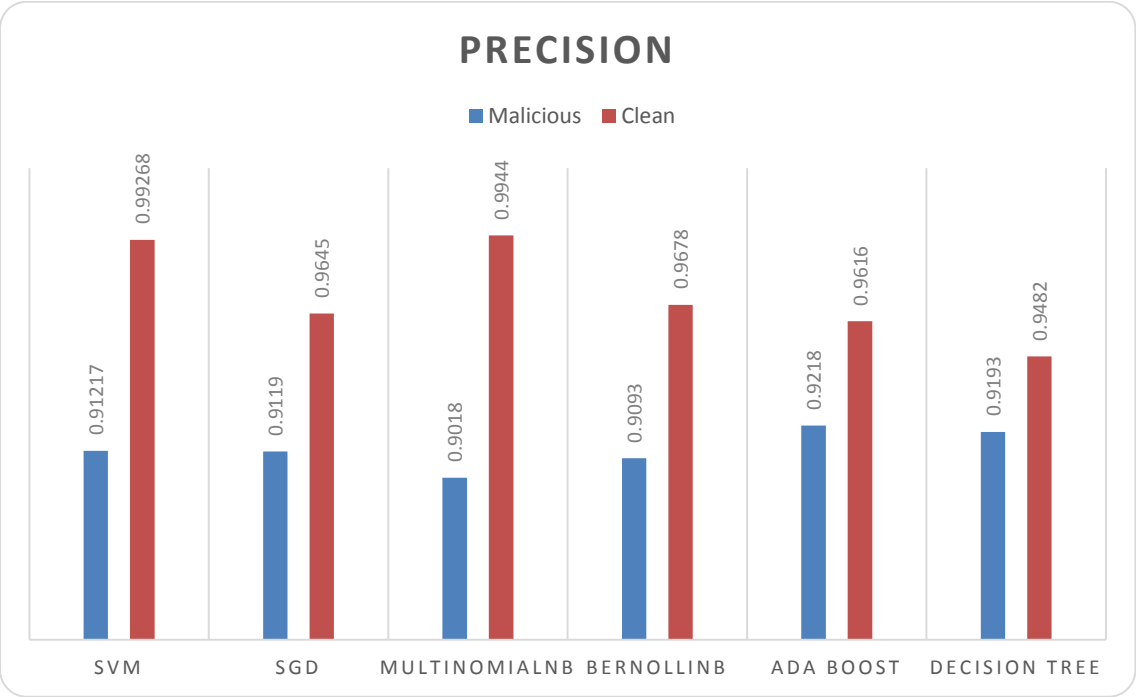

| | SVM | SGD | Multinomial NB | BernolliNB | Ada Boost | Decision Tree |
|---|---|---|---|---|---|---|
| Accuracy | 94.92% | 93.99% | 94.33% | 93.66% | 94.08% | 93.33% |
| Error rate | 5.08% | 6.01% | 5.67% | 6.34% | 5.92% | 6.67% |

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad (5.1)$$

Using equation ( 5.1) the confusion matrix accuracy, we can compute the accuracy of each classification algorithm. This evaluation evaluates the methods employed on training by representing how accurate is it and what is the ratio result. We found all the six methods have high accuracy with greater than 93% results.

Results in Table (5.4) gives us a good impression that our solution and features we selected to give best results against detecting polymorphic buffer overflow vector attacks. Precisely we can see that SVM is the best accuracy results with 94.9%. Computing error-rate for all of algorithms result that SVM has the smallest error rate with nearly 5%.

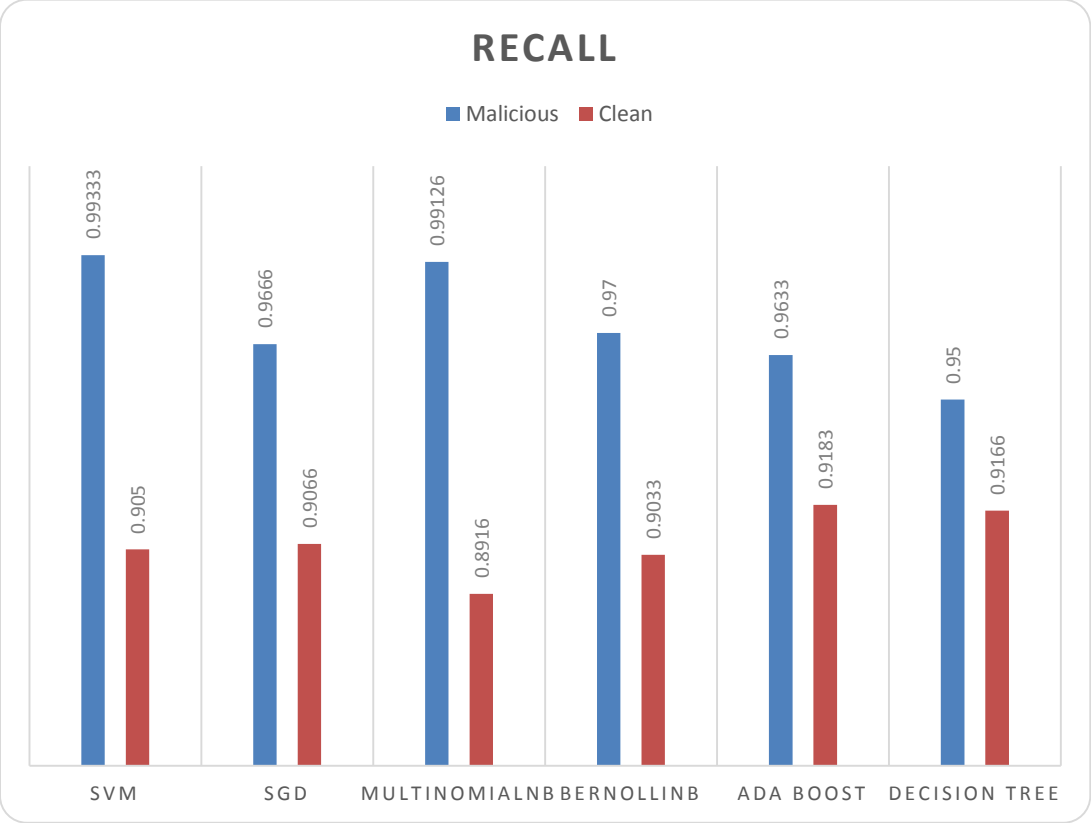
**Table (5.5):** Precision of experiments result




$$\text{Precision} = \frac{TP}{TP + FP} \qquad \textbf{(5.2 )}$$

Evaluating the malicious rate that correctly predicted by the system from overall system prediction computed using equation (5.2) (precision computation from confusion matrix). We can see in Table (5.5) the method of ADA BOOST has the highest value with 92% correct prediction precision and the others have greater than 90% in precision predicting the malicious label. On the other hand, we found that SVM and Multinomial Naïve Bayes have the highest rate with 99.2% with 99.4% respectively in the correct predicting precision of clean label. SVM predict eight clean from each 1000 files as malicious data which is a little bit small rate for the false alarm. Also, on Multinomial NB we have six false alarm from 1000 packets in our experiments.

**Table (5.6):** Recall of experiment results




$$\text{Recall} = \frac{TP}{TP + FN} \qquad \textbf{(5.3 )}$$

The rate of correctly malicious prediction from all of real malicious calculated from this equation (5.3) (Recall computation from confusion matrix). We can see in Table (5.6) that Decision Tree is the lowest rate in sensitivity detecting the malicious data with rate 95% and the other methods with a high rate greater than 96%. So we found that the engines models can sensitively identify most of the relevant malicious documents. This evaluation is critical, and we are using it as the first factor which results that SVM can detect 99.3% of the real malicious data as we take care to not miss any malicious packets in contrast of getting false alarm when there is no attack because I need to stop the real attack.

**Table (5.7):** F-Measure evaluation result from confusion matrix


$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \qquad \textbf{( 5.4 )}$$

We determined the overall performance quality of the classification algorithms by combine precision and recall in harmonic mean which computed by equation ( 5.4) (Compute f-measure evaluation metric from labeled precision and recall) we found that all the engines have excellent high rate results except Decision Tree as shown in Table (5.7).

## 5.3. Summary

From the security perspective, we need to choose the best classifier method, which produces results with the highest rate of correctly predicted from real overall results. That means we are looking for the sensitivity (Recall) of the algorithm, so we are depending on malicious RECALL evaluation in choosing the classifier method as the factor.

From experiments, which we have applied we got the evaluation results as shown in the charts. According to the factor which we depend on (the RECALL as we described), we found that SVM method scores high result with Recall rate is 99.3% in classifying the malicious packets. It fails on classifying clean data with ten samples in each 100 samples with a rate of 90.5%. This result means there is slight malicious can pass from SVM model, but we have a percentage of a false alarm which classifies clean data as malicious and this rate 9.5% so we see it can be acceptable rate and need to be improved in this stage as we care for not allowing any malicious packets to be pass.

To support our selection also SVM has the highest accuracy beside high precision. Also, the F-measure is very high.

Overall we choose SVM as the main classifier method for our solution according to the factor we chose as evaluation metrics.

# Chapter 6
# Conclusion and Future Work

In this research, we demonstrated how much is the buffer overflow is danger and how hackers can be employing the weapons of polymorphic Shellcodes to hack the systems and bypass security that can catch Shellcodes. Also, we mentioned about Intrusion detection systems and how they depend on signatures and how polymorphic Shellcodes can pass. We described other researchers solutions, which have different types of analysis trying to detect and prevent buffer overflow such as instruction frequency analyzer, or assuming packets is data and could not have instructions, or encrypt buffers with XOR, or by analyzing n-gram disassembly ..., etc.; we illustrated the defects in the related work and how can hackers bypass these solutions.

We worked on a new solution using data mining classification. This solution depends on the idea of getting the op-code of the CPU Intel architecture instruction sets for the polymorphic sled NOOPs of 32-bit and applying the classification on it. Only that can detect Buffer overflow polymorphic vector attack on network level before the Shellcode can conduct the victim host. Our solution depends on a self-generated dataset from Metasploit polymorphic NOOPs engines. We applied different classification algorithms on the dataset to get the perfect method that can deal with our problem. Solution experiments illustrated high accuracy in detecting malicious data on the network with low false alarm for most of the algorithms we used. We chose SVM as the best classification algorithm that can handle this issue because of it has 94% accuracy and getting 99.33% of malicious recall metrics and the low false alarm we get. Our solution shows significant results comparing against signature based on SNORT IDS which we compared against 1000 packets of polymorphic Shellcodes. By activating the latest Rules available on SNORT site. It can detect 502 packets of 1000 packets as a harmful packets with rate of 50.2%, on the other hand, our solution detects most of this packets with a near rate of 94% in this comparison experiment.

We are looking as future work to implement our solution as a plugin on SNORT IDS, to allow the solution work on the real environment. These will protect users and networks from the effectiveness of buffer overflow vulnerabilities. Also, we are looking to make the prediction of data type speeder with reducing the false alarm that system shows. Beside that we are looking to extract a new type of feature that can help in speeding the classification and give higher results in the evaluation.

# References

# The Reference List


Akritidis, P., Markatos, E. P., Polychronakis, M., & Anagnostakis, K. (2005, June 1). *STRIDE: Polymorphic Sled Detection Through Instruction Sequence Analysis.* Paper Presented at 20th International Information Security Conference Security and Privacy in the Age of Ubiquitous Computing IFIP TC11 (pp. 375-391). Chiba, Japan: Springer US.

Barwise, M. (2010, September 9). *What is an internet worm?* Retrieved March 5, 2016, from bbc: http://www.bbc.co.uk/webwise/guides/internet-worms

BeaEngine. (2013, May). *BeaEngine Sweet x86 x86-64 disassembler library*. Retrieved April 26, 2016, from BeaEngine Sweet: http://beatrix2004.free.fr/BeaEngine/index1.php

Bright, P. (2015, August 26). *How security flaws work: The buffer overflow*. Retrieved January 27, 2016, from http://arstechnica.com/security/2015/08/how-security-flaws-work-the-buffer-overflow/

Brownlee, J. (2014, April 16). *A Gentle Introduction to Scikit-Learn: A Python Machine Learning Library*. Retrieved from http://machinelearningmastery.com/a-gentle-introduction-to-scikit-learn-a-python-machine-learning-library/

*Buffer overflow*. (2016, January). Retrieved March 13, 2016, from Wikipedia: https://en.wikipedia.org/w/index.php?title=Buffer_overflow&oldid=737175721

Bulbapedia. (2016, March). *Arbitrary Code Execution*. Retrieved April 4, 2016, from Bulbapedia: http://bulbapedia.bulbagarden.net/wiki/Arbitrary_code_execution

Burns, B., Killion, D., Beauchesne, N., Moret, E., Sobrier, J., Lynn, M., . . . Granick, J. a. (2007). *Secure Power Tools.* O'REILLY.

Capstone. (2010, August). *Capstone The Ultimate Disassembler*. Retrieved April 28, 2016, from Capstone: http://www.capstone-engine.org

CLETteam. (2003). *Polymorphic Shellcode Engine Using Spectrum Analysis.* Retrieved January 2, 2016, from Phrack Inc: http://phrack.org/issues/61/9.html

*Computer Worm*. (2016, January). Retrieved March 1, 2016, from Wikipedia, The Free Encyclopedia.: https://en.wikipedia.org/w/index.php?title=Computer_worm&oldid=741766263

Cournapeau, D. (2007). *scikit-learn Machine Learning in Python*. Retrieved May 13, 2016, from scikit-learn: http://scikit-learn.org/stable/

DuPaul, N. (2013, December 3). *Static Testing vs. Dynamic Testing*. Retrieved March 1, 2016, from Veracode: https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing

Fen, T., Fuchao, Y., Xiaobing, S., Xinchun, Y., & Bing, M. (2012, March 19). *A New Data Randomization Method to Defend Buffer Overflow Attacks.* Paper Presented at International Conference on Applied Physics and Industrial Engineering. *24*(1), pp. 1757-1764. Nanjing ,China: ELSEVIER.

Gamayunov, D., Quan, N. T., Shakharov, F., & Toroshchin, E. (2009, Nov 10). *Racewalk: Fast Instruction Frequency Analysis and Classification for Shellcode Detection in Network Flow.* Paper Presented at Computer Network Defense (EC2ND). pp. 4-12. Mosco, Russia: IEEE.

Gushin, Y. (2008). *NIDS polymorphic evasion - The End?* Retrieved April 20, 2016, from ECL Labs: http://www.ecl-labs.org/papers/ecl-poly.txt

Han, J., & Kamber, M. (2005). *Data Mining: Concepts and Techniques* (2nd ed.). Peter Kriegel, Germany: Morgan Kaufmann.

Hsu, F.-H., Guo, F., & Chiueh, T.-c. (2006, Dec. 5). *Scalable network-based buffer overflow attack detection.* Paper Presented at ACM/IEEE Symposium on Architectures for Networking and Communications Systems (pp. 163-172). New York: IEEE.

Intel. (2003). *IA-32 Interl Architecture Software Developer's Manual: Basic Architecture* (1st ed.). Calefornia, USA: Intel.

K2. (2001). *a shellcode mutation engine.* Retrieved March 2, 2016, from ADMmutate: http://www.ktwo.ca/security.html

Khan, L., Thuraisingham, B., & Masud, M. (2011). *Data Mining Tools for Malware Detection.* (1st ed.). Texas: CRC Press.

Khodaverdi, J., & Farnaz, A. (2013, September). A Robust Behavior Modeling for Detecting Hard-coded Address Contained Shellcodes. *International Journal of Security and its Applications, 7*(5), pp. 101-112.

Masud, M., Khan, L., Thuraisingham, B., Wang, X., Liu, P., & Zhu, S. (2008). *Detecting Remote Exploits Using Data Mining.* Paper Presented at Digital Forensics Conference. *285*, pp. 177-189. California: Springer US.

National Institute Of Standards and Technology. (2014). *National Institute Of Standards and Technology*. Retrieved April 11, 2016, from National Vulnerability Database: https://nvd.nist.gov/home.cfm

Pasupulati, A., Coit, J., Levitt, K., Wu, S. F., Li, S. H., Kuo, J. C., & Fan, K. P. (2004, April 23). *Buttercup: on network-based detection of polymorphic buffer overflow vulnerabilities.* Paper Presented at Network Operations and Management Symposium. *1*, pp. 235-248. Seoul, South Korea: IEEE Xplore.

Polychronakis, M., Anagnostakis, K. G., & Markatos, E. P. (2006, July 13). Network–Level Polymorphic Shellcode Detection Using Emulation. *Detection of Intrusions and Malware & Vulnerability Assessment DIMVA 2006, 4064*(1), pp. 54-73.

Rapid7. (2004). *The Metasploit Project.* Retrieved May 1, 2016, from Metasploit: www.metasploit.org

Rapid7. (2013). *The ultimate guide to the Metasploit Framework*. Retrieved May 10, 2016, from Metasploit Unleashed: https://www.offensive-security.com/metasploit-unleashed/

SANS. (2002). *IDFAQ: What is polymorphic shell code and what can it do?* Retrieved January 20, 2016, from The SANS Institute: https://www.sans.org/security-resources/idfaq/what-is-polymorphic-shell-code-and-what-can-it-do/2/19

*Shellcode*. (2016). Retrieved March 9, 2016, from Wikipedia, The Free Encyclopedia.: https://en.wikipedia.org/w/index.php?title=Shellcode&oldid=741414732

Silberman, P., & Johnson, R. (2004, Aug. 04). *A Comparison of Buffer Overflow Prevention Implementations and Weaknesses.* New York: iDEFENSE Labs. Retrieved February 2, 2016, from http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf

SNORT. (2016). *Snort, the open-source network intrusion detection system*. Retrieved August 28, 2016, from SNORT: www.snort.org

Song, Y., Locasto, M. E., Stavrou, A., Keromytis, A. D., & Stolfo, S. J. (2009, October 29). On the infeasibility of modeling polymorphic shellcode Re-thinking the role of learning in intrusion detection systems. *Machine Learning, 81*(2), pp. 179-205.

Spafford, E. H. (1988). *The Internet Worm Program: An Analysis.* Indiana: Purdue University. Retrieved March 17, 2016, from http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1701&context=cstech

Sphinx. (2009). *Natural Language Toolkit Documentation*. Retrieved May 14, 2016, from Natural Language Toolkit: http://www.nltk.org

Symantec. (2016). *What is a Zero-Day Vulnerability?* Retrieved April 7, 2016, from PCTOOLS: http://www.pctools.com/security-news/zero-day-vulnerability/

Vento, D. D. (2016). *What is the difference between supervised learning and Unspervised learning.* Retrieved August 20, 2016, from Stackoverflow: http://stackoverflow.com/questions/1832076/what-is-the-difference-between-superv

Wang, W., Wang, H., Luo, D., & Fang, Y. (2007, October 16). *Online Detect Polymorphic Exploit Based on Data Mining.* Paper Presented at International Conference on Intelligent Systems and Knowledge Engineering (pp. 1435-1442). Chengdu,China: ISKE.

Wicherski, G., Cesare, S., & Carrera, E. (2016, May). *disassembly library*. Retrieved May 1, 2016, from Google Code: https://code.google.com/archive/p/libdasm/

Younan, Y. (2013). *25 Years of Vulnerabilityies: 1988- 2012.* Retrieved May 5, 2016, from Sourcefire Vulnerability Research Team (VRTTM):

https://labs.snort.org/blogfiles/Sourcefire-25-Years-of-Vulnerabilities-Research-Report.pdf

Yuan, J., & Ding, S. (2011, May 29). *A Method for detecting buffer Overflow Vulnerabilities.* Paper Presented at 3rd IEEE International Conference on Communication Software and Networks (pp. 188-192). Xi'an, China: IEEE.

Zaïane, O. R. (1999). *Principles of Knowledge Discovery in Databases*. Retrieved June 4, 2016, from https://webdocs.cs.ualberta.ca/~zaiane/courses/cmput690/

Zhao, Z., & Ahn, G.-J. (2013, Oct. 16). *Using instruction sequence abstraction for shellcode detection and attribution.* Paper Presented at 1st IEEE International Conference on Communications and Network Security, *CNS 2013* (pp. 323-331). Washington, DC, United States: IEEE Computer Society.

# Appendix (1)

msf_cmd_generate.py — UNREGISTERED

msf_cmd_generate.py

```
1  file_handler = open ("generator.rc","w")
2  file_handler.write("use nop/x86/opty2\n") # nop/x86/signle_byte for single byte
3  for file_no in range(1590,5000):
4      spool = "spool Desktop/nops/%s.nop\n" % (file_no)
5      file_handler.write(spool)
6      for try_no in range(1,100):
7          txt = "generate %s -t hex\n" % (file_no)
8          file_handler.write(txt)
9
10 file_handler.write("exit\n");
11 file_handler.close()
```

Line 1, Column 1 | Tab Size: 4 | Python

converter_to_asm.py — UNREGISTERED

converter_to_asm.py

```
1  import glob
2  import re
3  import os
4  from capstone import *
5  from binascii import unhexlify
6  r = re.compile("\[0m(.*?)resource")
7  path = 'c:/nops-5356-4999/'
8  for filename in glob.glob(os.path.join(path, '*.nop')):
9      #print filename
10     #break
11 #for filename in glob.glob('*.nop'):
12     with open(filename) as f:
13         lines = f.readlines()
14         for line in lines:
15             #print line
16             m = r.search(line)
17             if m:
18                 CODE = m.group(1)
19                 #print lines.index(line)
20                 CODE = unhexlify(CODE)
21                 md = Cs(CS_ARCH_X86, CS_MODE_32)
22                 head, tail = os.path.split(filename)
23                 new_file = 'c:/nop_asm/%s/%s'%(tail,lines.index(line))
24                 if not os.path.exists(os.path.dirname(new_file)):
25                     os.makedirs(os.path.dirname(new_file))
26                 file_handler = open(new_file,'w')
27                 for i in md.disasm(CODE, 0x1000):
28                     #print("0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
29                     #print(i.mnemonic)
30                     file_handler.write(i.mnemonic + '\n')
31                 file_handler.close()
32         #break
```

Line 32, Column 15 | Tab Size: 4 | Python

converte_clean_to_asm.py — UNREGISTERED

converte_clean_to_asm.py

```
1  import glob
2  import re
3  import os
4  from capstone import *
5  import binascii
6  import sys
7  path = 'c:/clean-data/'
8  for root, subFolders, files in os.walk(path):
9      for filename in files:
10         with open(os.path.join(root,filename),"rb") as f:
11             CODE = f.read()
12             try:
13                 CODE = binascii.hexlify(CODE)
14             except:
15                 print("##ERROR: %s " %sys.exc_info()[0])
16             md = Cs(CS_ARCH_X86, CS_MODE_32)
17             md.detail = True
18
19             head, tail = os.path.split(filename)
20             new_file = 'c:/clean_asm/%s.asm'%(tail)
21             if not os.path.exists(os.path.dirname(new_file)):
22                 os.makedirs(os.path.dirname(new_file))
23
24             file_handler = open(new_file,'w')
25             try:
26                 for i in md.disasm(CODE, 0x1000000):
27                     file_handler.write(i.mnemonic + '\n')
28             except CsError as e:
29                 print("ERROR: %s %s" %(e,f))
30             file_handler.close()
```

Line 30, Column 33 | Tab Size: 4 | Python