

# Identifying Malicious Code Through Reverse Engineering

# Advances in Information Security

---

**Sushil Jajodia**

*Consulting Editor*

*Center for Secure Information Systems*

*George Mason University*

*Fairfax, VA 22030-4444*

*email: [jajodia@gmu.edu](mailto:jajodia@gmu.edu)*

The goals of the Springer International Series on ADVANCES IN INFORMATION SECURITY are, one, to establish the state of the art of, and set the course for future research in information security and, two, to serve as a central reference source for advanced and timely topics in information security research and development. The scope of this series includes all aspects of computer and network security and related areas such as fault tolerance and software assurance.

ADVANCES IN INFORMATION SECURITY aims to publish thorough and cohesive overviews of specific topics in information security, as well as works that are larger in scope or that contain more detailed background information than can be accommodated in shorter survey articles. The series also serves as a forum for topics that may not have reached a level of maturity to warrant a comprehensive textbook treatment.

Researchers, as well as developers, are encouraged to contact Professor Sushil Jajodia with ideas for books under this series.

## ***Additional titles in the series:***

***SECURE MULTI-PARTY NON-REPUDIATION PROTOCOLS AND APPLICATIONS*** by José A. Onieva, Javier Lopez, Jianying Zhou; ISBN: 978-0-387-75629-5

***GLOBAL INITIATIVES TO SECURE CYBERSPACE: An Emerging Langscape*** edited by Michael Portnoy and Seymour Goodman; ISBN: 978-0-387-09763-3

***SECURE KEY ESTABLISHMENTS*** by Kim-Kwang Raymond Choo; ISBN: 978-0-387-87968-0

***SECURITY FOR TELECOMMUNICATIONS NETWORKS*** by Patrick Traynor, Patrick McDaniel and Thomas La Porta; ISBN: 978-0-387-72441-6

***INSIDER ATTACK AND CYBER SECURITY: Beyond the Hacker*** edited by Salvatore Stolfo, Steven M. Bellovin, Angelos D. Keromytis, Sara Sinclaire, Sean W. Smith; ISBN: 978-0-387-77321-6

***INTRUSION DETECTION SYSTEMS*** edited by Robert Di Pietro and Luigi V. Mancini; ISBN: 978-0-387-77265-3

***VULNERABILITY ANALYSIS AND DEFENSE FOR THE INTERNET*** edited by Abhishek Singh; ISBN: 978-0-387-74389-9

***BOTNET DETECTION: Countering the Largest Security Threat*** edited by Wenke Lee, Cliff Wang and David Dagon; ISBN: 978-0-387-68766-7

***PRIVACY-RESPECTING INTRUSION DETECTION*** by Ulrich Flegel; ISBN: 978-0-387-68254-9

***SYNCHRONIZING INTERNET PROTOCOL SECURITY (SIPSec)*** by Charles A. Shoniregun; ISBN: 978-0-387-32724-2

***SECURE DATA MANAGEMENT IN DECENTRALIZED SYSTEMS*** edited by Ting Yu and Sushil Jajodia; ISBN: 978-0-387-27694-6

For other titles published in this series, go to  
[www.springer.com/series/5576](http://www.springer.com/series/5576)

# Identifying Malicious Code Through Reverse Engineering

*edited by*

Abhishek Singh  
*Microsoft Corporation*  
*Redmond, WA, USA*

*with contributions by*

Baibhav Singh  
*Honeywell Technology Solutions Laboratory*  
*Bangalore, India*



*Editor:*

Abhishek Singh  
Microsoft Corporation  
One Microsoft Way  
Advanta-B/3099  
Redmond, WA 98052–6399, USA  
[abhishekhsingh243@gmail.com](mailto:abhishekhsingh243@gmail.com)

*with contributions by:*

Baibhav Singh  
Honeywell Technology Solutions Laboratory  
151/1, Doraisanipalya, Bannerghatta Road  
Bangalore – 560 076, India

ISBN: 978-0-387-09824-1  
DOI: 10.1007/978-0-387-89468-3

e-ISBN: 978-0-387-89468-3

Library of Congress Control Number: 2008942254

© Springer Science+Business Media, LLC 2009

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

[springer.com](http://springer.com)

## Preface

Vulnerabilities have increased since 2007. Vulnerability researchers find it difficult to get the source code of the software. Reverse engineering is one of the effective methods to analyze binaries for identifying vulnerabilities.

The book first discusses the assembly language. The chapter not only provides the fundamentals of assembly language. It also discusses about the various calling conventions and data constructs.

Since the programs are tightly coupled with the operating system, the second chapter discusses the fundamentals of operating system, about concepts of processes, threads, segmentation, context switching and the methods that can be used for synchronization between threads. Vista provides various security features such as ASLR, and pointer encoding which provides inherent protection against the vulnerabilities. The chapter also provides the details of new cryptographic API's in detail.

In chapter 3, PE file format, the executable and linking file format are discussed.

Chapter 4 discusses various vulnerabilities such as buffer overflow, format string vulnerability, SEH exception handler, Stack overflow, Off-by-One vulnerability, and Integer Overflow. The chapter first discusses the details of these vulnerabilities, using assembly code, The chapter also discusses the analysis of exploits for these vulnerabilities.

The last chapter focuses on the fundamentals of reverse engineering. It discusses the linear sweep disassembler, recursive disassembler, and various evasion techniques which can be used by the disassembler. The detection of hardware break point, software break point and the detection of virtual machine are also presented. The chapter concludes with the methods, which can be used to find the manual entry point of the executable and import table reconstruction.

The concepts discussed in the book are of practical use and the exploits are from the real world exploits. Although the book has been designed for those who practice information security, it can also be used for advanced information security level courses. The instructors can feel free to contact.

Abhishek Singh

# Table of Contents

## Assembly Language

1.0 Introduction .....	1
1.1 Registers .....	1
1.1.1 General Purpose Register .....	1
1.1.2 FLAGS Register .....	2
1.2 80x86 Instruction Format .....	3
1.2.1 Instruction Prefix .....	4
1.2.2 Lock and Repeat Prefixes .....	4
1.2.3 Segment Override Prefixes .....	4
1.2.4 Opcode .....	5
1.3 Instructions .....	7
1.3.1 Basic Instructions .....	7
1.3.2 Floating Point Instruction .....	10
1.4 Stack Setup .....	13
1.4.1. Passing Parameters in C to the Procedure .....	13
1.4.2 Local Data Space on the Stack .....	15
1.5 Calling Conventions .....	16
1.5.1 cdecl calling convention .....	16
1.5.2 fastcall calling convention .....	17
1.5.3 stdcall calling convention .....	17
1.5.4 thiscall .....	17
1.6 Data Constructs .....	17
1.6.1 Global Variables .....	18
1.6.2 Local Variables .....	18
1.6.3 Registers .....	19
1.6.4 Imported Variables .....	19
1.6.5 Thread Local Storage (TLS) .....	20
1.6.6 Executable Data Section .....	20
1.7 Representation of Arithmetic Operations in Assembly .....	21
1.7.1 Multiplication .....	22
1.7.2 Division .....	22
1.7.3 Modulo .....	24
1.8 Representation of Data Structure in Assembly .....	24
1.8.1 Representation of Array in Assembly .....	24
1.8.2 Representation of Linked List in Assembly .....	25
1.9 Virtual Function Call in Assembly .....	26
1.9.1 Representation of classes in Assembly .....	27
1.10 Conclusion .....	28

**Fundamental of Windows**

2.0 Introduction .....	29
2.1 Memory Management .....	29
2.1.1 Virtual Memory Management .....	29
2.1.1.1 Virtual Memory Management in Windows NT .....	32
2.1.1.2 Impact of Hooking .....	33
2.1.2 Segmented Memory Management .....	34
2.1.3 Paged Memory Management .....	36
2.2 Kernel Memory and User Memory .....	37
2.2.1 Kernel Memory Space .....	37
2.2.2 Section Object.....	38
2.3 Virtual Address Descriptor .....	39
2.3.1 User Mode Address Space .....	39
2.3.2 Memory Management in Windows .....	39
2.3.3 Objects and Handles .....	40
2.3.4 Named Objects .....	40
2.4 Processes and Threads .....	41
2.4.1 Context Switching .....	43
2.4.1.1 Context Switches and Mode Switches.....	43
2.4.2 Synchronization Objects .....	44
2.4.2.1 Critical Section .....	44
2.4.2.2 Mutex .....	44
2.4.2.3 Semaphore .....	45
2.4.2.4 Event .....	45
2.4.2.5 Metered Section .....	45
2.5 Process Initialization Sequence .....	46
2.5.1 Application Programming Interface .....	47
2.6 Reversing Windows NT .....	48
2.6.1 ExpEchoPoolCalls .....	49
2.6.2 ObpShowAllocAndFree .....	49
2.6.3 LpcpTraceMessages .....	49
2.6.4 MmDebug .....	49
2.6.5 NtGlobalFlag .....	49
2.6.6 SepDumpSD .....	50
2.6.7 CmLogLevel and CmLogSelect .....	50
2.7 Security Features in Vista .....	50
2.7.1 Address Space Layout Randomization (ASLR) .....	50
2.7.2 Stack Randomization .....	51
2.7.3 Heap Defenses .....	52
2.7.4 NX .....	54
2.7.5 /GS .....	55
2.7.6 Pointer Encoding .....	56
2.7.7 Cryptographic API in Windows Vista .....	58

2.7.8 Crypto-Agility .....	59
2.7.9 CryptoAgility in CNG .....	60
2.7.10 Algorithm Providers .....	62
2.7.11 Random Number Generator .....	63
2.7.12 Hash Functions .....	64
2.7.13 Symmetric Encryption .....	65
2.7.14 Asymmetric Encryption .....	67
2.7.15 Signatures and Verification .....	68
2.8 Conclusion .....	68

## **Portable Executable File Format**

3.0 Introduction .....	69
3.1 PE file Format .....	69
3.2 Import Address Table .....	77
3.3 Executable and Linking Format .....	79
3.3.1 ELF Header .....	79
3.3.2 The Program Header Table .....	80
3.4 Conclusion .....	83

## **Reversing Binaries for Identifying Vulnerabilities**

4.0 Introduction .....	85
4.1 Stack Overflow .....	85
4.1.1 CAN-2002-1123 Microsoft SQL Server 'Hello' Authentication Buffer Overflow" .....	88
4.1.2 CAN -2004-0399 Exim Buffer Overflow .....	88
4.1.3 Stack Checking .....	90
4.2 Off-by-One Overflow .....	90
4.2.1 OpenBSD 2.7 FTP Daemon Off-by-One .....	93
4.2.3 Non-Executable Memory .....	94
4.3 Heap Overflows .....	94
4.3.1 Heap Based Overflows .....	96
4.4 Integer Overflows .....	106
4.4.1 Types Integer Overflow .....	108
4.4.2 CAN-2004-0417 CVS Max dotdot Protocol Command Integer Overflow .....	111
4.5 Format String .....	112
4.5.1. Format String Vulnerability .....	113
4.5.2 Format String Denial of Service Attack .....	115
4.5.3 Format String Vulnerability Reading Attack .....	115
4.6 SEH Structure Exception Handler .....	116
4.6.1 Exploiting the SEH .....	119
4.7 Writing Exploits General Concepts .....	122
4.7.1 Stack Overflow Exploits .....	122

4.7.2 Injection Techniques .....	123
4.7.3 Optimizing the Injection Vector .....	123
4.8 The Location of the Payload .....	123
4.8.1 Direct Jump (Guessing Offsets).....	124
4.8.2 Blind Return .....	124
4.8.3 Pop Return .....	124
4.8.4 No Operation Sled .....	125
4.8.5 Call Register .....	125
4.8.6 Push Return .....	126
4.8.7 Calculating Offset .....	126
4.9 Conclusion .....	126
<b>Fundamental of Reverse Engineering</b>	
5.0 Introduction .....	127
5.1 Anti-Reversing Method .....	127
5.2.1 Anti Disassembly .....	128
5.2.1.1 Linear Sweep Disassembler.....	128
5.2.1.2 Recursive Traversal Disassembler.....	130
5.2.1.3 Evasion of Disassemble .....	131
5.2.2 Self Modifying Code .....	135
5.2.3 Virtual Machine Obfuscation .....	139
5.3 Anti Debugging Techniques .....	140
5.3.1 BreakPoints.....	142
5.3.1.1 Software Breakpoint .....	142
5.3.1.2 Hardware Breakpoint.....	143
5.3.1.3 Detecting Hardware BreakPoint .....	144
5.4 Virtual Machine Detection .....	145
5.4.1 Checking Fingerprint Inside Memory, File System and Registry ..	145
5.4.2 Checking System Tables .....	145
5.4.3 Checking Processor Instruction Set .....	146
5.5 Unpacking .....	147
5.5.1 Manual Unpacking of Software .....	148
5.5.1.1 Finding an Original Entry Point of an Executable.....	148
5.5.1.2 Taking Memory Dump .....	154
5.5.1.3 Import Table Reconstruction .....	156
5.5.1.4 Import Redirection and Code emulation .....	162
5.6 Conclusion .....	166
<b>Appendix .....</b>	<b>168</b>
<b>Index .....</b>	<b>187</b>

# Assembly Language

## 1.0 Introduction

Assembly language implements a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. Ollydbg (available at <http://www.ollydbg.de/>) or IDA pro (available at <http://www.hex-rays.com/idapro/>) are the two most commonly used tools used to disassemble binary to extract assembly instructions from machine level language. Operations of software are visible in the assembly language. Understanding of assembly language is required to get a better understanding of the low level software binaries. This chapter focuses on assembly for 32-bit Intel Architecture (IA-32)

## 1.1 Registers

IA-32 has various registers. We can categorize them according to their usage as general purpose register, segment register, index register, instruction pointer register and status registers. In addition, there are seven other registers used for debugging any application. Break points can be applied through these registers. The letter "E" in the name of registers indicates that these registers have been extended from the 16 bit Intel Architecture.

### 1.1.1 General Purpose Register

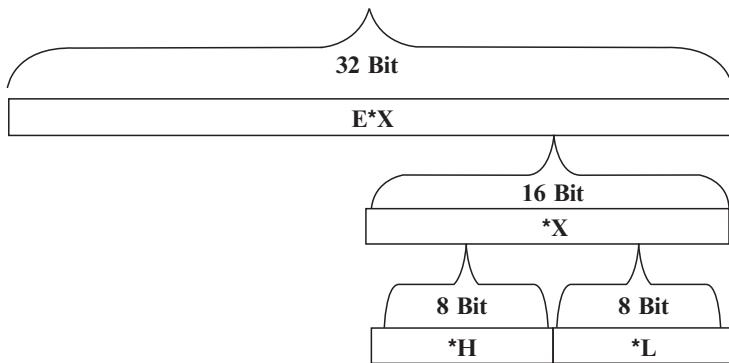
Segment registers are part of the x86 Segment Memory Model. The length of these registers is 16 bit. These registers point to the memory segment. In X86 there are following segment registers.

**CS (Code Segment registers)** - points to the code segment of an application.

**DS (Data segment Registers)** - points to the data segment of the application.

**FE/FS/GS (Auxiliary segment registers)** - These are extra segment registers used for pointing address

**SS (Stack Segment Register)** - points to the stack segment of an application. Its value is same as that of DS sometimes.



### 1.1.2 FLAGS register

The FLAGS registers are the status register that stores the information about the status of the processor. Different bits denote different status of the registers

?	NT	IO	OF	DF	IF	TP	SF	ZF	?	AF	?	PF	?	CF
15	14	13	12	11	10	←						0		

Some common flags bits their description is given below.

CF → Carry Flag gets sets if the last operation has generated a carry.

? → Reserved by Intel for future use.

PF → Parity Flag indicates if the number of set bits is odd or even in the binary representation of the result of the last operation

ZF → Zero Flag gets set to 1 if the result of last instruction is zero

SF → Sign flag gets sets if the result of the last operation was negative.

TP → This Flag is set when instruction is debugged set by step.

IF → If interrupt flag is set, it denotes CPU will handle hardware interrupts.

DF → This flag is used by string processing instruction.

OF → Overflow Flag gets set if overflow is triggered by the last arithmetic operation

IO → Two bits 12 and 13 in the FLAGS register denotes IO flag. This shows the I/O privilege level of the current program or task.

Register	Usage of Register
EAX, EBX, EDX	They are used for integer, Boolean, logical or memory operations.
ESI/EDI	Used as counter by repetitive instruction that requires counting
EBP	Used as generic register. It is also used as a stack based pointer. Stack based pointer is used to create stack frame. Local and the parameters passed to the function are accessed by stack frame. The base pointer EBP points to the stack position. This address is right after the return address for the current function.
ESP	ESP is a CPU's stack pointer. Current position in stack is stored in ESP register. This register gets updated when anything is pushed to the stack. Anything which is pushed to the stack is stored in address in ESP.

Figure 1.0 showing various registers and their usage.

Besides the registers mentioned in figure1.0, IA-32 has a special register called EFLAGS. This register comprises status and system flags. Status flags which are updated by logical and integer instructions contain the current logical state.

## 1.2 80x86 Instruction Format

The figure 1.1 shows the general instruction format. The instruction of 1A-32 is a subset of general instruction format. From the figure 1.1 it can be concluded that the length of the instruction can be upto 16 bytes, but 80x86 doesn't allow instruction greater than 15 bytes.

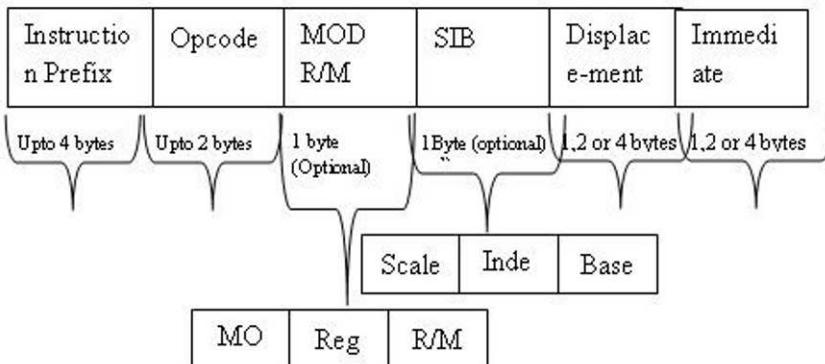


Figure 1.1 Showing the General Instruction Format

### 1.2.1 Instruction prefix

Instruction prefix can be upto 4 bytes that define the behavior of the instruction. These instruction prefixes can be divided into following four groups. The behavior of various prefix bytes are mutually exclusive and results are undefined if these mutually exclusive prefixes are added in front of an instruction.

### 1.2.2 Lock and repeat prefixes:

Lock prefix (0xF0) forces the operations to be atomic that can be used to access exclusive shared memory in multiprocessor environment. The repeat prefixes (0xF2 – REPNE/REP NZ and 0xF3 – REPE/REP Z) makes an instruction to be repeated for each element of string. This prefix can only be used with the string operations.

### 1.2.3 Segment override prefixes

Operand-size override prefixes (0x66) denote the size of the operand which allows a program to switch between 16 to 32 bit operand. If the instruction

doesn't contain this prefix then operand size will be of default size; else this prefix will select non default size.

Address-size override prefixes (0x67) denote the addressing mode, which allows the program to switch between 16 to 32 bit addressing. If the instruction doesn't contain this prefix, then address size will be of default size, else this prefix will select non default address size.

#### 1.2.4 Opcode

80x86 supports two Opcode size :one byte standard opcode and two- byte opcode. The two-byte opcode instruction is prefixed with 0x0F opcode expansion byte. The second byte in the two-byte opcode specifies the actual instruction. Sometimes an additional 3-byte op code field is encoded in ModR/M byte.

Various instruction classes use few bits of the opcode as a sign and direction flag. The Zero th bit of the opcode specifies the size of the operand. If this bit contains one then the operands are either 16-bits or 32-bits. Under 32-bit operating systems, the default is 32-bit operands if this field contains a one. Bit number one is the direction bit which identifies the direction of the transfer. If this bit is zero, then the destination operand is a memory location. If this bit is one, then the destination operand is a register.

For example: -

#### MODR/M AND SIB BYTES

REG specifies any of the eight register of the 80x86. This register can be either source or destination. This can be determined with the help of d flag present in the opcode field. The operand is the source if d ==0 and it is destination id d==1. But for various single operand instruction the REG field may contain an opcode extension rather than a register value.

The MOD and R/M fields specify the other operand in a two-operand instruction. The following table specifies how the MOD and reg fields together specify the addressing modes

There are two displacing modes -- 8 bit and 32 bit. 8 bit addressing mode displacement exists for a displacement in between the range of -128 to 127. As these displacements can be mentioned through one byte the instruction

will be shorter as compared to 32 bit instruction. At most of the places these shorter instructions are found to save lot of space.

In addition, there is one more addressing mode called scaled indexed addressing mode. This represents addressing mode of the form [ebx + edx \* 4]. The table shown in figure 1.3 explains the mode of addressing.

<b>Index</b>	<b>Register</b>
%000	EAX
%001	ECX
%010	EDX
%011	EBX
%100	Illegal
%101	EBP
%110	ESI
%111	EDI

<b>Index * Scale value</b>	<b>Scale value</b>
Index * 1	% 00
Index * 2	%01
Index * 4	% 10
Index * 8	% 11

<b>Base</b>	<b>Register</b>
% 000	EAX
% 001	ECX
% 010	EDX
% 011	EBX
%100	ESP
%101	Displacement if MOD= %00, EBP if MOD = %01 or %10
%110	ESI
%111	EDI

**Figure 1.3 showing modes of addressing**

## 1.3 Instructions

In IA-32, instructions comprise op code which will be followed by operand. Op codes are the assembly instructions whereas operands are the parameters to the instructions. Operands work in three ways by using registers, immediate or by using the memory address. When registers are used to access data, they are stored in general purpose registers. In the immediate method, the constant value is embedded in the code. This also indicates that a hard coded value is used in the original program. When data resides in RAM then a memory address is used to access them. Memory addresses are enclosed in brackets denoting value at address. These addresses can either be hardcoded or the address can be stored in registers. Registers can also be used to store the base address along with a constant which represents an offset into that object.

### 1.3.1 Basic Instructions

Some of the most commonly used instructions are discussed below.

- **test *arg1, arg2*** : Test instruction is used to perform bit-wise AND on the two operands. However it has to be noted that it does not store the result.

The flags that the test instruction modifies are as follows:

- Carry flag
- Overflow flag
- Parity Flag
- Sign Flag
- Zero flag

- **cmp *arg1, arg2*** : cmp instruction performs subtraction between the two operands. However it does not store the result. If the result of comparison is zero, the Zero Flag (ZF) is set.

The flags that test the cmp instruction modifies are as follows:

- Carry flag
- Auxiliary flag
- Overflow flag
- Parity Flag
- Sign Flag
- Zero flag

- **jmp *loc***: The instruction will load the EIP with the specified address.

- **je loc:** The instruction will load EIP with the specified address if the operands of previous cmp instructions are equal.  
The jump condition: jump if Zero Flag == 1
- **jne loc :** This instruction will load EIP with the specified address. This will happen when the operands of previous CMP instructions are not equal.  
The jump condition: jump if Zero Flag == 0
- **jg loc:** If the first operand of the previous CMP instruction is greater than the second then the EIP is loaded with the specified address.  
The jump condition: jump if ZF=0 and SF=OF
- **jge loc:** If the first operand of the previous CMP instruction is greater than or equal to the second then the EIP is loaded with the specified address.  
SF=OF
- **ja loc:** The instruction will load the EIP with the specified address. This will happen when the first operand of the previous CMP is greater than the second.  
CF=0 and ZF=0
- **jae loc:** This will load the EIP with the specified address if the first operand of the previous CMP is greater than or equal to the second  
jae, is the same as jge.  
CF=0
- **jl loc:** The instruction “jl” represents jump if less than, this gets executed or EIP is loaded with the specified address when the first operand of the previous CMP is less than the second operand  
SF != OF
- **jle loc:** The instruction “jle” represents jump if less than or equal to , this gets executed or the EIP is loaded with the specified address when the first operand of the previous CMP is less than or equal to then the second operand.  
ZF=1 or SF != OF
- **jo loc:** This instruction loads the EIP with the specified instruction if the overflow bit is set on a previous arithmetic expression.  
OF=1
- **jnz loc:** This will load the EIP with the specified address. This will happen then the zero bit is set from a previous arithmetic expression.  
ZF=1
- **jz loc:** This operation will load the EIP with the specified address. This will happen if the zero bit is set from a previous arithmetic expression. This expression is identical to the je.

ZF=0

- call *proc* : This operation is mostly used when subroutines are called and will push EIP +4 onto the top of stack. After this the instruction will jump to the specified location.
- ret [*val*] : This instruction will load the next value on the stack into the EIP and then it will pop the stack the specified number of times. The instruction will not pop any values off the stack if the field “val” is not specified.
- loop *arg* : This instruction decrements ECX. It will jump to the address specified by arg. Besides loop other instructions which decrement the ECX counter are loope, loopne, loopnz, loopx.
- enter *arg* : This instruction allocates space on the stack and creates a stack frame.
- leave: The instruction will destroy the current stack frame and restore the previous frame.
- hlt : This instruction will halt the processor.
- nop: This instruction does nothing and wastes an instruction cycle. It is converted into XCHG operation with the operands EAX and EAX.
- wait: This instruction waits for the CPU to finish its last calculation.
- mov *arg1 arg2*: The “mov” instruction takes in two operands. The destination operand which can be a memory address or the register and source operands which can be an immediate, register or a memory address. It moves the date from arg2 to arg1 (from source to destination operand.)
- ADD *arg1 arg2*: The add instruction adds unsigned or signed integers storing the result in arg1.

The flag that tests the cmp instruction modifies are as follows:

- Carry flag
- Auxiliary flag
- Overflow flag
- Parity Flag
- Sign Flag
- Zero flag

- SUB *arg1 arg2*: The sub instruction subtracts the value of arg2 from arg1 and stores the value in arg1. The instruction is again valid for both signed and unsigned integers.

The flags that test the cmp instruction modifies are as follows:

- Carry flag
- Auxiliary flag
- Overflow flag

- Piraty Flag
  - Sign Flag
  - Zero flag
- MUL *arg*: The instruction will multiply the unsigned operand by EAX. The result of the multiplication is stored in a 64-bit EDX:EAX. The low 32 bits are stored in EAX and the high 32 bits are stored in EDX.
  - DIV *arg* : The instruction divides the 64 bit unsigned value stored in EDX:EAX by the unsigned *arg*. The quotient is stored in EAX and the remainder is stored in EDX.
  - IMUL *arg*: By using the instruction, the signed operand is multiplied by the EAX and the result is stored in EDX:EAX.
  - IDIV *arg*: The instruction divides the 64-bit value stored in EDX:EAX by the signed operand storing the quotient in EAX and the remainder in EDX.
  - SHR Op *arg* : It shifts the number stored in Op to the *arg* number of the bit to the right.
  - SHL Op *arg* : It shifts the number stored in Op to the *arg* number of the bits to the left.
  - CDQ : The instruction zero and extends the value. The instruction sign extends an eight bit value to 32 or 64 bits.
  - movsx : It copies the content of source to destination. Sign extends the value. The extended value is dependent upon the operand-size attributed. .

### 1.3.2 Floating point instruction

These floating point instructions are executed by the x87 coprocessor. On encountering any floating point instruction, the x86 processor communicates the instruction to x87. At the same time x86 instructions keep on executing other instructions until and unless it encounters another floating point instruction or next instruction require the result of executing floating point instruction. In that case WAIT instruction is executed to halt the execution of x86 processor. There are various compilers that emulate the x87 instruction. This emulation is done through interrupt. Linker replaces the original floating point instruction with the interrupt instruction. On the occurrence of these interrupt, the interrupt handler function is executed that interprets and emulate these instructions.

There are 8 floating point registers; the name of these instructions are from ST(0) to ST(7). These are not real registers, but stack is used for this purpose. Each register occupies 10 bytes. In addition to these 8 registers, there is a register of 14 bytes for status and control information.

The floating point instruction can be classified into various categories -- data movement instruction, conversion, arithmetic instruction, comparison constant instructions, transcendental instructions, and miscellaneous instructions.

The data movement instructions transfer data between the internal FPU registers and memory. The instructions in this category are fld, fst, fstp, and fxch. The fld converts 32 and 64 bit operand to an 80 bit extended precision value and pushes it onto the floating point stack. The FLD instruction first decrements the top of stack pointer that is denoted by bits 11-13 of the status register and then stores the 80 bit value in the physical register specified by the new TOS pointer. Just opposite to it is FSTP instruction that always pops the top of stack. The FST and FSTP instructions copy the value on the top of the floating point register stack to another floating point register or to a 32, 64, or 80 bit memory variable. The floating point rounding control bit is referred to the 80 bit extended precision value on the top of stack is rounded to the smaller format (32, 64, or 80 bit memory variable). The FXCH instruction exchanges the value on the top of stack with one of the other FPU registers. There are two variants of FXCH instruction, one with a single FPU register as an operand and the other with without any operands. If operand is mentioned then this instruction exchanges the top of stack (tos) with the specified register. If no operand case FXCH instruction swaps the value at top of stack with ST1.

Various floating point instructions are available to compare real values. The instructions such as FCOM, FCOMP and FCOMPP compare the two values that are present on top of the stack. In case of floating point instruction, there are no conditional jump instructions. To test the condition, the FSTSW and SAHF instructions can be used. The FSTSW instruction copies the floating point status register to the AX register and SAHF instruction copies the AH register into the 80x86's condition code bits. After that normal x86 conditional jump instructions can be used to test condition. The FCOM, FCOMP, and FCOMPP instructions either compare ST0 to the specified operand. or compare ST0 against ST1 if no operand is specified and set the processor flags accordingly. If the operand is 32 or 64 bit memory variable then it is first converted into 80-bit extended precision value and then compare ST0 against this value. FCOMP pops the ST0 after the comparison.

As most of the floating point instruction requires with 32 or 64 bit memory variable to convert the value into 80 bit extended precision value before performing any operation and then it perform the require operation on these variable. There are few FPU instructions that convert to or from integer or binary coded decimal (BCD) format. For example, FILD instruction converts a 16, 32, or 64 bit two's complement integer to the 80 bit extended precision format and pushes the result onto the stack. Then, the floating point operation can be done on this value.

Floating point instruction also supports arithmetic instruction set to perform arithmetic operations. Few of the common instructions include FADD, FSUBB ,FMUL, FDIV etc. These are lot of variant instructions available in each category for example Add arithmetic instruction has following variants

fadd

faddp

Pop the two values from the stack, addition is performed on them and the result is pushed back to the stack

fadd st(i), st(0)

fadd st(0), st(i)

It is same as that of x86 ADD instruction, the value in the second register operand gets added to the value in the first register operand. Here either of the two register operands must be st(0)

faddp st(i), st(0) here st(0) must always be the second operand and its value is added to the second register operand and then st(0) is popped.

fadd mem

Here the operand is a 32 or 64 bit memory operand. This instruction will convert the 32 or 64 bit operands to an 80 bit extended precision value and then will add the value in st(0).

These above instructions contain various types of operands, these are

1. Floating point stack that can be denoted as ST(i) where i can be 0 to 7
2. 10-byte memory operand containing a full precision floating point value.
3. 8-byte memory operand containing a double precision floating
4. 4-byte memory operand containing a single precision floating point
5. 10-byte operand containing a special Binary Coded Decimal format
6. 4-byte operand representing a signed integer in two's-complement notation.

7. 2-byte operand representing a signed integer in two's-complement notation.

## 1.4 Stack Setup

Setting up of a Stack frame is required before entering a procedure. This stack frame will be required to pass the parameters. The stack set up can be identified by the following assembly code in binary.

```
push    ebp  
mov     ebp, esp
```

The first instruction push ebp saves the value of register EBP into the stack. Here EBP contains the address of the last stack frame created. Here the value of Ebp is saved as this value will be required after the completion of the execution of the routine as the control will be returned to the called function and its stack form is needed in order to access local variables and parameter. The second instruction is moving the current stack pointer value to EBP register. The current stack pointer value is moved to the EBP as further the local and the parameter will be referred with EBP register.

EBP allows the use of a pointer as an index into the stack. It should not be altered throughout the procedure. Each parameter passed to the procedure can be accessed as an offset from EBP. This is known as a "standard stack frame."

\$ 55	PUSH EBP
• 89E5	MOV EBP,ESP
• 89EC 0C	SUB ESP,0C
• B9 03000000	MOV ECX,3

Figure 2.0 Showing setting up of a Stack Frame

The procedure should preserve the content of the register ESI, EDI, EBP and all the segment registers. An error will be generated if these registers are corrupted. As shown in figure 2.0 these instructions are used for stack setup.

### 1.4.1 Passing Parameters in C to the Procedure

C passes arguments to procedures on the stack. For example, consider the following statements from a C main program:

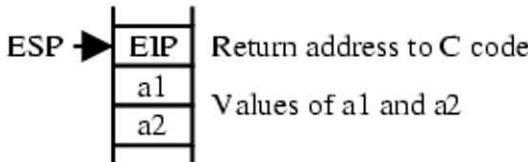
```
#include<stdio.h>
#include<string.h>

int add_int(int a1, int a2){
    int c;
    c= a1 + a2;
    return c;
}

void main(){
    add_int(10,20);
}
```

Figure 3.0 showing C code for adding two integers

When C executes the function call to `add_int`, it pushes the input arguments onto the stack in *reverse* order, then executes a call to `add_int`. Upon entering `add_int`, the stack would contain the following:



The method of passing parameters shown in figure 3.0 is called passing by value. The variables `a1` and `a2` are declared as `int` variables, each takes up one word on the stack. The code for `Sum`, which outputs the sum of the input arguments via register `EAX`, looks similar to that shown in figure 4.0

<pre>\$ 55            PUSH EBP . 89E5          MOV EBP,ESP . 51            PUSH ECX . B9 01000000   MOV ECX,1 &gt; 49            DEC ECX . C7048C 5A5AF0   MOU DWORD PTR SS:[ESP+ECX*4],FFF5A5A5 .^75 F6          JNZ SHORT testchar.004012DD . 57            PUSH EDI . 8B7D 08          MOU EDI,DWORD PTR SS:[EBP+8] . 037D 0C          ADD EDI,DWORD PTR SS:[EBP+C] . 897D FC          MOV DWORD PTR SS:[EBP-41],EDI . 8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4] . 5F            POP EDI . C9            LEAVE . C3            RETN \$ 55            PUSH EBP . 89E5          MOV EBP,ESP . 6A 14          PUSH 14 . 6A 0A          PUSH 0A . E8 D1FFFFFF   CALL testchar._mul . 83C4 08          ADD ESP,8 . C9            LEAVE . C3            RETN</pre>
---

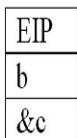
Figure 4.0 Assembly instructions for code shown in figure 3.0

As shown in figure 4.0, the instructions push EBP, Mov EBP; ESP denotes the initialization of a stack frame. The value is returned to the C code by using EAX implicitly. RETN is used for returning from a procedure. This is due to the fact that C takes care of removing the passed parameters from the stack. As it can be seen in the above example, only one output value is returned.

It might also happen that the value is passed by reference. For example the function

```
add_int(b1,&b2);
```

The first argument is still passed by value (i.e., only its value is placed on the stack), but the second argument is passed by reference (its *address* is placed on the stack).



In this case, it has to be noted that the &c is pushed on the stack, not its value. EAX is the only register which can be used by the assembly to return values to the C calling program. In case the return value is less than 4 bytes, the result is returned in the EAX register. If the return value is larger than 4 bytes, then the pointer is returned in EAX.

A short table of the C variable types and how they are returned by the assembly code:

Register Containing Return Value	Data Type
AL	Char
AX	Short
EAX	int long pointer(*)

Figure 5.0 C variables returned by assembly code.

### 1.4.2 Local Data Space on the Stack

As shown in figure 3.0, the variable C is a local variable. By subtracting the value form ESP that is current stack point, temporary storage space is allocated in the stack for local variables. The space on the stack is below the return address and the base pointer. Since in stack frame EBP points to that, the assembly code which requires access to the variable can use

EBP and subtract offset from it. As shown in figure 6.0, the variable C shown in the figure is represented by the highlighted part. As shown in figure 6.0, for the instruction

`MOV DWORD PTR SS:[EBP-4],EDI`

since the local variable is accessed using a fixed hardcoded offset, so it can be assumed that the local variable is of the fixed size. Once the procedure is executed it is also important to restore the stack space. This is done by adding the value that is subtracted from the register ESP in the start of the function and by restoring the register value that has been restored in the stack.

```

PUSH EBP
MOV EBP,ESP
PUSH ECX
MOV ECX,1
DEC ECX
AI MOV DWORD PTR SS:[ESP+ECX*4],FFFFA5A5
JNZ SHORT testchar.0040120D
PUSH ESI
PUSH EDI
MOV EDI,DWORD PTR SS:[EBP+8]
MOV ESI,DWORD PTR SS:[EBP+C]
ADD EDI,DWORD PTR DS:[ESI]
MOV DWORD PTR SS:[EBP-4],EDI
MOV EAX,DWORD PTR SS:[EBP-4]
POP EDI
POP ESI
LEAVE
RETN
    
```

Figure 6.0 Assembly code for C code show in figure 3.0

## 1.5 Calling Conventions

Calling conventions define how the functions are called in a program. They decide the arrangement of data in a stack when a function call is made. In the below mentioned sections some of the common calling convention are discussed.

### 1.5.1 cdecl calling convention

The cdecl calling convention permits functions to receive a dynamic number of parameters. The calling convention receives the parameters in a reverse order with the first parameter pushed on to the top of the stack first and the last parameter pushed last. In this calling convention, it is the responsibility of the caller to restore the stack pointer after the execution of the called function. As this category of function might have variable number of argument and so stack pointer can only be restored by callee function. A

function which takes one or more parameters and ends with a simple RET with no operands is the cdecl function.

### 1.5.2 fastcall calling convention

This calling convention makes use of registers for passing the first two parameters passed to a function. It makes use of registers ECX and EDX to store the first and second parameters respectively. The remaining parameters are passed through stack. Fastcall calling convention increases the execution speed on the procedure as application accesses register rather than stack value

### 1.5.3 stdcall calling convention

This is mostly used in windows. The argument passing method and the order are opposite to the cdecl calling convention. In stdcall calle function is responsible for clearing its own stack. However in cdecl functions, it is the responsibility of the caller to clear the function stack. The stdcall function uses the RET instruction for clearing the stack. It can receive operands which specify the number of bytes to be cleared from the stack after jumping from the stack. The operand passed to RET exposes the number of bytes passed as a parameter. The operand has to be divided by four to get the number of parameters.

### 1.5.4 thiscall

This is used by the C++ function call with a fixed number of parameters. For this function call, a valid pointer is loaded in ECX, and the parameters are pushed onto stack without using EDX as a valid C++ method function call. If there are a dynamic number of parameters then the compiler will use cdecl and pass *this* pointer as the first parameter.

## 1.6 Data Constructs

This section presents the representation of data constructs by compiler in low level assembly language. During reversing, this knowledge can help to identify the data constructs in an assembly language.

### 1.6.1 Global Variables

Global variables are initialized by the system when they are defined. They reside in a fixed memory address in an executable. As shown in figure 7.0, variable *d* is a global variable.

```
#include<stdio.h>
#include<string.h>

int d= 10;

int mul_int(int a1, int a2){
    int c;
    c = a1 * (a2) * d;
    return c;
}

void main(){
    int b = 10;
    int c = 20;
    mul_int(b,c);
}
```

Figure 7.0 Showing C code with a global variable.

When they are accessed, hardcoded addresses are used. This makes it easier to spot the global variables in binary. As shown in figure 8.0, hard coded address “FFFA5A5A” is being used to access the global variable. The hardcoded address is mostly used by compilers for global variables.

```
PUSH ELO
PUSH EAX
MOV ECX,2
DEC ECX
MOV DWORD PTR SS:[ESP+ECX*4],FFFA5A5A
JNZ SHORT testchar.00401309
MOV DWORD PTR SS:[EBP-4],0A
MOV DWORD PTR SS:[EBP-8],14
PUSH DWORD PTR SS:[EBP-8]
PUSH DWORD PTR SS:[EBP-4]
```

Figure 8.0 showing the Assembly for global variable in figure 7.0

### 1.6.2 Local Variable

They are used by functions to store immediate values. These values can either be stored in a stack or they can be stored in a register. For example, as shown in figure 7.0 for the function *mul\_int*, *c* is a local variable. Storing local variables in stack has been discussed in detail in the section Stack Setup.

When the parameter area of the stack is written by the function, then it can be inferred that the space is being used to hold some extra variables. A function rarely returns value to the caller by writing parameters back to the parameter area of the stack. Call by reference is used when parameters passed by the called function is modified and again used by the calling function.

### 1.6.3 Registers

Registers are generally used to store the immediate value. They generate the fastest code. Many compilers have various optimization techniques which aid in generating optimized code. The variables which are used most extensively are placed in registers. The “volatile” key word indicates that the variable will be read and written asynchronously by the software and the hardware. So the local variables which are declared as “volatile” are always accessed by using the memory address. The “register” keyword indicates to the compiler that it is a heavily used variable and should be placed in registers. However, it may happen that the compiler will follow its own optimization algorithm and can ignore the keyword “register”. Hence, for the keyword “register” there is no distinguishable mark in the assembly code.

### 1.6.4 Imported Variables

They are global variables which are stored and maintained in another binary module. For being able to successfully export a module, the exporting and the importing module must both refer to the same variable name. It might happen that the variable is exported by ordinals, so the variable is not named. Since an imported variable involves an additional level of redirection, identifying them is a simple process. The assembly for identifying the imported variable is similar to that shown in figure 9.0

```
mov eax, DWORD PTR [Import Address Table Address]  
mov ebx, DWORD PTR [eax]
```

The above mention code reads data from a pointer which in itself points to another pointer. Here it has to be noted that the value is the value of the Import Address Table Address. Hence any double pointer redirection, where the first pointer is addressed to the Import Address Table is the reference to the import variable.

A constant variable can be defined by using the #define directive. When a #define directive is used, then the value is replaced in the preprocessing stage.

Another method to define a constant variable it to define a global variable and add the *const* keyword to the definition. This generates the code as if it is a regular global variable. The enforcement of the *const* keyword is done by the compiler. Some compilers can arrange the global variables in two sections, one which is read only and another which is both readable and writeable. The constants will be placed in the read only section.

### 1.6.5 Thread Local Storage (TLS)

TLS are generally used for managing thread specific data structures. One of the methods to implement thread local storage programs is to use the TLS API. The TLS API includes various functions TlsAlloc, TlsGetValue and TlsSetValue. These API's provide programs with the ability to manage a small pool of thread local 32 bit value. Another approach can be to define a global variable with the *declspec(thread)* attribute which places the variable in a thread- local section of the image executable. For such cases the variable can be identified as a thread local since the variable points to a different image section than the rest of the global variables in the executables.

### 1.6.6 Executable Data Section

The executable data section is used to store the application data. This area is generally used to store either the preinitialized data or global variables.

```
#include<stdio.h>
#include<string.h>

int d= 10;
char* testglobalstring ="Executable's preinitialized data section
will be used to store the global string";

int mul_int(int a1, int a2){
    int c;
    char* testlocalstring =" Executable's preinitialized data
    section will be used to store the local string";
    c = a1 * (a2) * d;
    printf("%s\n",testglobalstring);
    printf("%s\b",testlocalstring);
    return c;
}

void main(){
    int b = 10;
    int c = 20;
    mul_int(b,c);
}
```

Figure 9.0 Showing C code having preinitialized data in local and global variables.

Preinitialized data comprises hard-coded values or constant data inside the program. As shown in figure 9.0, `testlocalstring` and `testglobalstring` contain preinitialized data. Some preinitialized data can be stored inside the code; however when the size of data is too large, the compiler stores them inside special areas in the program executable and generates code that references it by address.

Figure 10.0 showing the executable data section containing preinitialized data in the executable data section.

As shown in the figure 10.0, the *testlocalvariable* is a local variable; however it is still stored inside the preinitialized data section. As shown in figure 10.0, the *testglobalstring* which is a global string is also stored inside the executable data section. For global variables the value of the variables is retained through out the program which can be accessed any where from the program. With the preinitialized data, a hardcoded memory address is used to access the global variables. Hence besides an overlong value, another case where the data is stored inside the executable data section is for the global variables.

## 1.7 Representation of Arithmetic Operations in Assembly

The section discusses basic arithmetic operations and their implementation by various compilers. Even though the IA-32 processor provides instructions for multiplication and division, they can be slow. Hence it might be implemented in different ways in a compiler. The SHL instruction can be used to shift the values to the left which is the same as multiplying by the power of 2. Similarly the SHR can be used to shift the value to the right which is equivalent to dividing by the power of two. Multiplication and division compilers can use SHL, SHR instructions and then use addition and subtraction to compensate for the result.

### 1.7.1 Multiplication

Generally when a variable is multiplied by another variable MUL/IMUL is used. As shown for the code in 9.0, the multiplication instruction is shown in figure 11.0

```
| LEA EDI,DWORD PTR DS:[40A0B0]
MOV DWORD PTR SS:[EBP-8],EDI
MOV EDI,DWORD PTR SS:[EBP+8]
IMUL EDI,DWORD PTR SS:[EBP+C]
IMUL EDI,DWORD PTR DS:[d]
MOV DWORD PTR SS:[EBP-4],EDI
```

Figure 11.0 showing the multiplication of variables in figure 9.0 for the function mul\_int.

However, instead of using IMUL or MUL, other instructions can be used as well. For example, multiplying a number by three, is usually implemented by shifting a number by 1 bit and adding the original value to the result. This is done by using SHL and ADD or it can be done by using LEA.

```
lea eax, DWORD PTR [eax+eax*2]
```

### 1.7.2 Division

The instructions DIV and IDIV are being used for division. They have latency of around 50 clock cycles.

```
int div_int(int a1, int a2){
    int c;
    c = a1/3;
    return c;
}

void main(){
    int b = 10;
    int c = 20;
    div_int(b,c);
}
```

Figure 12.0 C code showing division by 3.

As shown in figure 13.0 the division is identified by the IDIV operation. Even though the operation is slow, it can easily be identified by reversing.

```

MOV DWORD PTR SS:[ESP+ECX*4],FFFFA5A5A
JNZ SHORT testchar.004012DD
MOU EAX,DWORD PTR SS:[EBP+8]
MOV ECX,3
CDQ
IDIV ECX
MOV DWORD PTR SS:[EBP-4],EAX
MOV EAX,DWORD PTR SS:[EBP-4]
LEAVE
RFTN

```

Figure 13.0 showing the assembly for the division shown in code mentioned in figure 12.0

It might happen that the compiler can use an efficient division technique. One of the methods is *reciprocal multiplication* which is an optimized division technique. Reciprocal multiplication is based upon the concept of using multiplication instead of division to implement division. It has to be noted that multiplication is four to six times faster on the IA-32 processor. The basic concept in reciprocal multiplication is to multiply the dividend by a fraction which is the reciprocal of the divisor. For example, to divide  $x/y$ , compute  $1/y$  and multiply it with  $x$ . As the data type is represented only in integer the fixed point arithmetic is used. It provides representation of fraction and real numbers without a decimal point.

Figure 14.0 shows some of the 32-bit reciprocals used by the compilers. These reciprocals are used along with the divisor which is the power of two.

Divisor	Reciprocal value	32-bit representation of reciprocal	Divisor in source Code
2	2/3	0xAAAAAAAB	3
2	4/5	0xCCCCCCCCD	5
4	2/3	0xAAAAAAAAB	6

Figure 14.0 showing the some of the cases for reciprocal multiplication.

For a divisor with a power of two only right shifts are required. These instructions help in achieving greater accuracy. In assembly instructions, reciprocal multiplication is easy to identify.

```

mov ecx, eax
mov eax, 0xffffffff
mul ecx
shr edx, 4
move eax, edx

```

Figure 14.1 Assembly code showing division by 5

The above shown code multiples the value in ecx with 0xFFFFFFFF, then it shifts the value by four. The combination of division and multiplication is equivalent to the divisor by five.

### 1.7.3 Modulo

To calculate modulo, division has to be performed; however a different part of the result is required.

```
mov eax, DWORD PTR [Divisor]
cdq
mov edi, 10
idiv edi
```

Figure 14.2 Assembly code for Modulo

The code shown in the figure divided the divisor by 10, then it places the result in EDX. The instruction idiv is used to perform a signed division instruction. It places the result of the division in EAX and the remainder in EDX. The instruction cdq, converts 64 bit dividend in EDX:EAX.

## 1.8 Representation of Data Structure in Assembly

Data structure is represented by a chunk of memory which represents a collection of different type of fields. The arrangement which is of static size is defined during compile time. It is also possible to create data structures in which the last member is a variable size array and the code for the structure is allocated dynamically at run time. Since the stack is of fixed size, for such type of structure, the stack is not allocated. Compilers usually align the structure to the processor's word size. This alignment to the processor's size will happen even if the structure is not of the word size. For example, even though Boolean uses one bit of storage, the compiler will allocate 32 bits of storage space.

### 1.8.1 Representation of Array in Assembly

An array is defined as a list of data structures stored sequentially in the memory. In assembly, the array access can be identified as the compiler in the assembly instruction will use some variable, to the object's base address. As shown in figure 15.0, the function array\_int initializes an array of size integers. For the initialization of the array, the equivalent instructions in assembly is shown in figure 16.0. The highlighted instruction, “*DWORD PTR SS:[EBP+EDI\*4-2C]*” is an access to array by using the base pointer.

```

#include<stdio.h>
#include<string.h>

int array_int(int a1, int a2){
    int c[10];
    int i=0;
    for (i=0;i<10;i++)
        c[i]=a1 + a2;
    return c[0];
}

void main(){
    int b = 10;
    int c = 20;
    array_int(b,c);
}

```

Figure 15.0 showing the array of integers.

It might happen that the array contains some hard coded addresses in the high level language. In such cases it will be difficult to identify the assembly instruction of array from the assembly instruction of any other data structure.

```

10V DWORD PTR SS:[EBP-4],0
10V DWORD PTR SS:[EBP-4],0
10V EDI,DWORD PTR SS:[EBP-4]
10V ESI,DWORD PTR SS:[EBP+8]
3DD ESI,DWORD PTR SS:[EBP+C]
10U DWORD PTR SS:[EBP+EDI*4-2C],ESI
INC DWORD PTR SS:[EBP-4]
CMP DWORD PTR SS:[EBP-4],0A
JL SHORT testchar.004012F9
10V EAX,DWORD PTR SS:[EBP-2C]

```

Figure 16.0 showing the assembly for array of integers.

Arrays are often accessed sequentially and like other data structures they are not aligned by the compilers. Array of pointers, integers or single word sized items consist of generic data structure. For the generic data type array, the index is simply multiplied by the machines word size. For a 32-bit processor, it resolves to multiply by four. To access the desired memory address, the desired memory index is multiplied by four and the result should be added to the array's starting address. As shown in figure 16.0 *DWORD PTR SS:[EBP+EDI\*4-2C]*, is used to access the memory address in the array. EDI stores the index of the memory and *[EBP-2C]* is the array's starting address. The data structure array is similar to the conventional array with the difference being that the item size can be of any value.

## 1.8.2 Representation of Linked List in Assembly

Linked lists are used when the items are generally added or removed from different parts of the list. Unlike arrays the items stored in the link list cannot be directly accessed through their index. In the linked list, the items

are scattered in the memory, and each item contains a pointer to the next item. In case of a double list it will contain a pointer to the previous item as well. In the case of an array, the items are stored sequentially. In the case of a single link list the data structure contains a combination of payload and pointer to the next. The pointer next points to the next item.

<pre> PUSH EBP MOV EBP,ESP SUB ESP,0C MOV ECX,3 DEC ECX MOV DWORD PTR SS:[ESP+ECX*4],FFFA5A5A JNZ SHORT linklist.0040120F PUSH EDI PUSH 0C CALL &lt;JMP.&amp;CRTDLL.malloc&gt; ADD ESP,4 MOV DWORD PTR SS:[EBP-C],EAX PUSH 0C PUSH 0 PUSH DWORD PTR SS:[EBP-C] CALL &lt;JMP.&amp;CRTDLL.memset&gt; ADD ESP,0C MOV DWORD PTR SS:[EBP-8],0 CMP DWORD PTR DS:[_LL_serv],0 JNZ SHORT linklist.00401342 PUSH 0C CALL &lt;JMP.&amp;CRTDLL.malloc&gt; ADD ESP,4 MOV DWORD PTR DS:[_LL_serv],EAX PUSH 0C PUSH 0 </pre>	<span style="color: red;">[size = C (12.) malloc]</span> <span style="color: red;">[n = C (12.) c = 00 s = memset]</span> <span style="color: red;">[size = C (12.) malloc]</span> <span style="color: red;">[n = C (12.) c = 00]</span>
--	---

## 1.9 Virtual Function Call in Assembly

Assembly code in Figure 19.0 shows the implementation of virtual function call. It has to be noted that the CALL does not use a hard coded address but is accessing data structure to get function's call. ECX register is used here for the address. This indicates that the function pointer resides inside the object instances, which are an indicator of the virtual function call. For the code shown in Figure 19.0, it also can be inferred that the function takes in no parameter.

```

mov eax, DWORD PTR [edi]
mov ecx, edi
call DWORD PTR [eax+4]

```

In assembly, for INTEL and Microsoft compiler, any function call, which loads a valid pointer into ECX, and indirectly calls a function whose address is obtained via the same pointer, is a C++ virtual member function call. For other compilers it might be tough because they do not use ECX for passing *this* pointer. Constructors perform the initialization of virtual function table pointers for inherited objects. For two constructors--one for base class

and another for its inherited class--both of them initialize the object's virtual function table. The base class sets the virtual function pointer to its own copy. This gets replaced, by the inherited class constructor, upon return of the function call.

### 1.9.1 Representation of classes in Assembly

Classes in C++ contain a combination of data and code which operates on them. This section discusses analyzing binary to analyze the classes in C++. A class with no inheritance is similar to the data structure with associated function. “*this*” pointer, which is used as an instance of class, is typically passed via ECX register. The assembly code of accessing plain data structure will be identical to the assembly code when plain data structure is accessed. Figure 18.0 shows the inherited class memory layout.

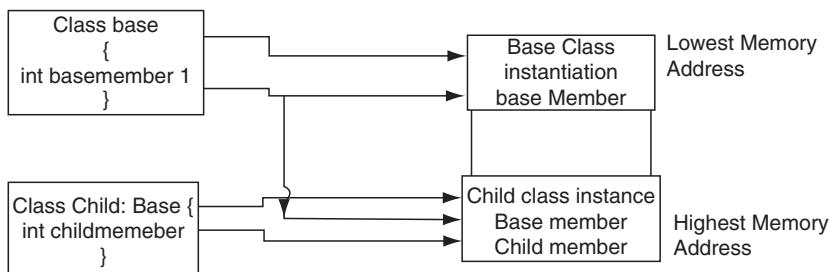


Figure 18.0 Memory address layout for the class methods.

A non-virtual function call can be considered as a direct function call. For this function call, this pointer is passed as the first argument. Some compilers such as G++ push this pointer on to the stack where as other compilers such as Intel’s and Microsoft compiler use ECX register to access the this pointer.

Virtual functions are implemented by the use of virtual function table. Virtual function table is placed in the .rdata, the read-only data section of executable. They contain hard-coded pointer to all the function implementations in a class. This pointer in turn aid in finding the correct function when call to these methods is made. Virtual function table are created at compile time for the classes that define virtual function and for the classes that are descendants and provide overload implementation of virtual function defined in the other class. VFTABLE pointer is added by the compiler during the runtime. During object instantiation, the VFTABLE pointer is initialized to the correct virtual function table.

## 1.10 Conclusion

This chapter presented the concepts of assembly language and description of the commonly used assembly language instructions. It discussed the stack setup and arrangement of local and global variables inside the stack. Cdecl, fastcall , stdcall, this call are some of the ways by which functions are called in a program. The chapter covered the local, global and imported variables and their identification in assembly. Multiplication, Division and modulo are some of the arithmetic operations. The chapter discussed methods of identifying these arithmetic operations. The chapter concluded by discussing various data structures and their implementation in assembly. The chapter has covered low level topics that are required for reverse engineering process.

# Fundamental of Windows

## 2.0 Introduction

Programs are tightly coupled with the operating system. So for reversing of binaries it becomes important to understand the principles and features of operating system. Some of the features which are discussed in this chapter are, virtual memory, portability, multithread, multiprocessor capability, security and compatibility. Windows NT is a 32-bit computing environment however, the current operating system also support 64-bit versions. Windows NT was a combination of C and C++, so it can be recompiled to run on different processors. It is also a fully pre-emptive multithreaded system. Windows NT also provides support for multiprocessor capability. This makes Windows NT suited for high performance computing. In Windows NT every object has an access control list, which determines which users are allowed to manipulate the objects. In terms of security, Vista provides Address Space layout randomization (ASLR). ASLR involves randomly arranging the positions of key data areas. This includes the base of the executable and position of libraries, heap, and stack, in a process's address space. ASLR is effective in prevention against the buffer overflow exploits. Access Control List is provided for each file for windows NT. It supports encryption for each file. Windows NT is compatible with the older version of applications executing on a 16-bit platform. The chapter discusses these points in detail.

## 2.1 Memory Management

One of the most important parts of operating system is memory management. Virtual memory is one of the solutions used for limited memory. It increases the memory of computer system by sharing the memory with the process.

### 2.1.1 Virtual Memory Management

Whenever CPU needs data or executable program, it brings them into memory. This is quite similar to the instructions and data when they are brought into the cache. One of the ways to control the memory management is

by using combination of hardware memory controller along with the operating system. Memory management is implemented using virtual memory. By using virtual memory, *each process* appears to have available the full memory resources of the system. Even though processes occupy the same virtual memory, they will be mapped into completely different physical memory area.

The part of program and data which are being executed lie in the main memory. *Virtual address translation* is used for translation from physical memory address to the data in the virtual memory address. Figure 1.0 shows the relationship between the name variable and physical location.

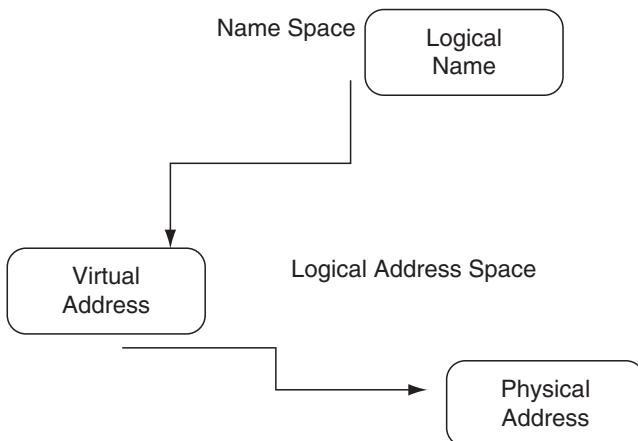


Figure 1.0 The name space to physical address mapping

The method to achieve the mapping is quite similar to the mapping main memory to cache memory. It has to be noted that in the case of virtual address mapping the relative speed of main memory to disk memory is high. This is approximately 10,000 to 100,000. Therefore, cost of miss in main memory is very high. In many processors direct mapping scheme is supported. Under this scheme, a *page map* is maintained in physical memory. Each physical memory reference requires both an access to page table and an operand. Most of the memory references are indirect. Virtual to physical address mapping is shown in Figure 2.0

Direct mapping from virtual to physical address will result in a considerable performance penalty. This is avoided in most of the systems by using *translation lookaside buffer* (TLB). TLB contains last few addresses and their physical addresses. Hence, in most of the cases, virtual to physical memory address does not require additional memory address. A typical virtual-to-

physical address mapping in a system containing a TLB is shown in Figure 3.0.

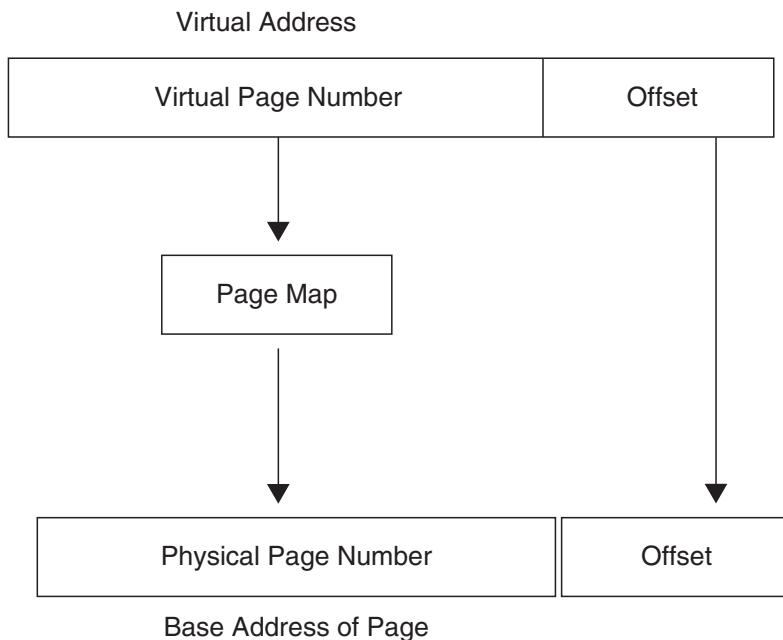


Figure 2.0 A direct mapped virtual to physical address translation

If the addresses are 32 bits in size then the size of virtual address space is  $2^{32}$  bytes or 4 GB. Disk contains the sections of program and data which are not executed normally. It might happen that the virtual memory refers to a location which is not in physical memory. In such a case the execution of that instruction is aborted. It can be restored when the required information is placed in the main memory from the disk controller. The processor can be executing another program in the meantime. The time to find the program is not wasted by the processor. The time required to place the information in memory can affect the time a user must wait for the result. A processor might have to wait if many disk-seeks are required. *Segmentation* and *Paging* are two of the size methods that can be used for memory management. In Segmentation memory management the memory is in segments and in the case of Paging memory management, the memory is in pages.

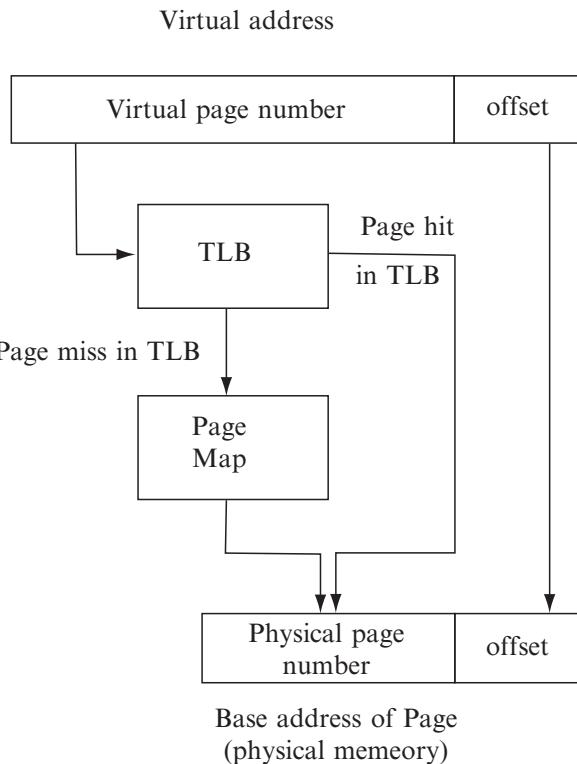


Figure 3.0 virtual to physical address translation mechanism with a TLB

#### *2.1.1.1 Virtual Memory Management in Windows NT*

First-in first-out replacement policy is used by windows. The oldest data is thrown out whenever there is a space crunch. In Windows NT, address space is broken down into 4KB pages and it maintains the information in the page table entry (PTE). The structure of PTE is processor dependent. The page is marked as invalid if the page is not mapped to the physical RAM. When the page contains DLL code or executable module code, the page is brought in from the Swap file. Windows NT keeps a track of free physical RAM in Page Frame Data Base (PFD). This ensures the allocation of space in case of page fault. Before discarding a page, Windows NT ensures that the page is not dirty. If the page is dirty, the page is written to the secondary storage before it can be written to the secondary storage. If the page is not shared, the PFD contains the pointer to PTE. In the case the page is shared, the PFD contains pointer to the corresponding PROTOPTE entry. In this case the PFD also contains a reference count for the page.

If the reference count for a page is 0, then it is discarded. PFD is an array of 24-byte entry, one for each physical page. Hence, the size of page is equal to the number of physical pages stored in the kernel variable `MmNumberOfPhysicalPages`. The kernel variable `MmpfnDatabase` contains pointer to the array. There can be different states to a physical page. For example, the physical page can be free, in use, free but dirty. PFD entry is linked in a double-linked list depending on the physical page represented by it. Depending upon the state of physical page, PFD entry is linked in a double-linked list, that is, if the PFD entry is representing use pages, it is linked to the use pages list. In sum, there are six kinds of list. The heads of these list are stored in the `MmStandbyPageListHead`, `MmStandbyPageListHead`, `MmModifiedNoWritePageListHead`, `MmModifiedPageListHead`, `MmFreePageListHead`, `MmBadPageListHead`, `MmZeroedPageListHead` kernel variables. These heads are 16 bytes each. The definition of the head is defined as follows.

```
typedef struct PageListHead {
    DWORD NumberOfPagesInList,
    DWORD TypeOfList,
    DWORD FirstPage,
    DWORD LastPage
} PageListHead_t;
```

The `FirstPage` field can be used as an index into the PFD which contains pointer to the next page. PFD entry has the following structure definition.

```
typedef struct PfdEntry {
    DWORD NextPage,
    void *PteEntry/*PpteEntry,
    DWORD PrevPage,
    DWORD PteReferenceCount,
    void *OriginalPte,
    DWORD Flags;
} PfdEntry_t
```

### *2.1.1.2 Impact of Hooking*

DLL codes are shared by all process and is write-protected. Hence, a process cannot alter the code of a DLL. However, it is possible to hook to a DLL in Windows NT. The first few bytes of a function need to be changed for hooking in the function call. Hence, for hooking the attribute of a page containing DLL code to read-write then the code bytes are altered. However, when the page is altered, a separate copy of the page is made and the write went to that page. This ensures that all the original pages use the unaltered page. Windows NT uses one of the available PTE bits for distinguishing pages which are marked as read-write and read-only.

Copy-on-write mechanism is used by Windows NT for various purposes. The DLL data pages are shared with the copy-on-write purposes which ensure that whenever a process writes to a data page, it makes a copy of the page. Other processes use the original copy of the pages. Location of DLL can be at different linear address for different processes. Depending upon the linear address where DLL is loaded, DLL need to be adjusted. The process is called as relocating the process. Windows NT marks the DLL code pages which are relocating as copy-on-write. This ensures that the pages requiring page relocation are copied per processes. Pages that do not have memory references in them are shared by all processes. Hence it is recommended that DLL has a preferred base address and loaded at the address. By specifying a base address, it can be ensured that the DLL need not be relocated. Hence if all processes load the DLL at preferred base address, they share the same copy of DLL code.

Copy-on-write mechanism in the Windows NT is used by the POSIX subsystem for implementing the fork system call. The fork system call is being used to create a child process of calling process. The child process shares the same state of code and data pages as the parent process. Since these are two different processes, the data pages should not be shared by them. The child process-invoked exec system call, which discards the current memory image of the processes, loads a new executable module and starts executing the new module. The fork-system call, marks the data pages as copy-on-write to prevent the copying of data pages. Data pages are copied only if the parent or the child writes to it. Copy-on-write is used to attain the efficiency in Windows NT memory management.

### 2.1.2 Segmented memory management

The blocks in a segmented memory management system that are to be replaced in main memory are potentially of unequal length and correspond to program and data "segments." The former segment could be, for example, a subroutine or a procedure, whereas the latter segment could be either a data structure or an array. In both cases, segments correspond to logical blocks of code or data. Therefore, segments are "atomic," because either the whole segment or none of the segments should be present in main memory. Although the segments may be placed anywhere in main memory, it is imperative that the instructions or data in one segment be contiguous, as shown in Figure 4.0

Segment 1
Segment 4
Segment 5
Segment 7
Segment 8

Figure 4.0 A segmented memory organization

Using segmented memory management, the memory controller needs to know the location of the start and the end of each segment in physical memory. When segments are replaced, a single segment can only be replaced by a segment of the same size, or by a smaller segment. After a time this results in a "memory fragmentation", with many small segments residing in memory, having small gaps between them. Because the probability that two adjacent segments can be replaced simultaneously is quite low, large segments may not get a chance to be placed in memory very often. In systems with segmented memory management, segments are often "pushed together" occasionally to limit the amount of fragmentation and allow large segments to be loaded.

While using segmented memory management, it is mandatory that the memory controller knows the location of the start and the end of each segment in physical memory. In the case when segments are to be replaced, a single segment can be replaced only by another segment that is of either the same size or a smaller size. After a while, such segment replacements can lead to "memory fragmentation," where many small segments reside in memory separated by small gaps. Given the low probability for two adjacent segments to be replaced at the same time, it is often not possible for large segments to be placed in memory. To limit the amount of fragmentation and allow large segments to be loaded, the segments in segmented memory management are often "pushed together" occasionally.

This is an efficient organization since not only an entire block of code is available to the processor but also it is also easy for two processes to share the same code in a segmented memory system. Hence there needs to be a single copy of the code. Majority of the current processors support a hybrid of paged memory management and segmented memory management, in which the segments consist of multiples of fixed-size blocks.

### 2.1.3 Paged memory management

In paged memory management, all of the segments are exactly the same size (typically 256 bytes to 16 K bytes). Virtual ``pages'' in auxiliary storage (disk) are mapped into fixed page-sized blocks of main memory with predetermined page boundaries. The pages do not necessarily correspond to complete functional blocks or data elements, as is the case with segmented memory management. The physical address of the new page in memory has to be determined, since the pages are not stored in contiguous memory location. Page Translation table is used to determine the address of new page. Page Translation uses associative memory to determine the physical address of the new page in the main memory. If the page is not found in the main memory then the CPU is interrupted then the page is requested from disk controller and the execution starts on another process.

Many other attributes are also usually included in a PTT. This is done by adding extra fields to the table. Pages or segments may be characterized as read-only, read-write. Moreover, it is common to include information about access privileges to help ensure that one program does not inadvertently corrupt the data of another program. The ``dirty'' bit indicates whether a page has been written to, so that the page will be written back onto the disk if a memory write has occurred onto that page. It is unusual to map all of main memory using associative memory because the latter is very expensive; therefore, only the physical addresses of recently accessed pages are maintained in a small amount of associative memory and the remaining pages in physical memory are maintained as a ``virtual address translation table'' in main memory. If the virtual address is contained in the associative memory then translation from virtual to physical address can be done in one memory cycle. If the physical address must be recovered from the ``virtual address translation table'' in main memory, at least one more memory cycle is needed. Trade-off exists between the page size for a system and the size of the PTT since if a processor has a small page size, the PTT must be large enough to map all of the virtual memory space. The paged memory management system provides inherent advantage over the segmented one is that the memory controller, which is required to implement a paged memory management system, is considerably simpler. In addition, the paged memory management does not suffer from fragmentation as does segmented memory management, although another kind of fragmentation does occur. A whole page is swapped in or out of memory, even if it is not full of data or instructions. Fragmentation is within a page in a paged memory management. It does not persist in the main memory when new pages are swapped in. when a large number of processes are executed ``simultaneously'' similar to a multiuser

system, the main memory may contain only a few pages for each process, and all processes may have only enough code and data in main memory to execute for a very short time before a page fault occurs. This situation, often called “thrashing,” degrades the throughput of the processor severely because it actually must spend time waiting for information to be read from or written to the disk.

## 2.2 Kernel Memory and User Memory

Memory management requires distinction between the kernel and the user memory space. Differentiation between the user and the kernel space prevents the bugs from user memory to be overwritten to the kernel space. It also prevents malicious software in the user space from taking control of the operating system.

Windows uses 4 GB of address space. Out of these 4 GB, 2 GB is used by the application memory space and the remaining 2 GB by the kernel address space. The upper 2GB of kernel space is protected from being used by the program.

### 2.2.1. Kernel Memory Space

The 2 GB of the kernel code contains various components such as device divers and the like. Figure 5.0 shows the layout for the windows kernel space. Physical memory and various user configurable registry keys determine the size of components which are allocated during the run time. Paged and in paged pool in the kernel space comprises all the kernel mode components. They are stored in the entire kernel mode. Caching is implemented in windows by mapping files into memory and allowing system cache is the place where windows cache manager stores all the currently cached file. When a program later access file, using ReadFile or WriteFile API, the system file internally access the mapped copy of the file cache manager API such as CcCopyRead and CcCopyWrite. Terminal Services Session Space component of the kernel mode component is used in WIN32K.SYS permits for multiple, remote GUI sessions on a single windows system. This memory space is made, “session private”. This enables to load the multiple instances of the win32 subsystem. As shown in the Figure 5.0 page table and the hyper space comprises the process-specific data which defines the current processes’ address space. System working set, comprises the system global data structure which manages the system’s physical memory.

Kernel Code (0x80000000)
Non paged Pool 12Mb (0x80DA6000)
(0x819A6000)
Additional System PTEs
(0xBE000000)
Terminal Services Session Space 32Mb
(0xC0000000)
Page Table (Process -Private)
(0xC0400000)
Hyper Space (process - Private)
(0xC0800000)
(0xC0C00000)
System working Set 4Mb
(0xC1000000)
System Cache Space 512Mb
(0xE1000000)
Paged pool 192 Mb
(0xED000000)
System PTEs 200 Mb
(0xF96A8000)
Extra Non Paged Pool

Figure 5.0 Showing Windows Kernel Space.

System-page Table Entries is a large virtual memory space which can be used by the kernel and the drivers. A system page table entry which comprises of virtual memory space is used for kernel allocation which is used by the kernel and drivers. Device drivers make use of `MmAllocateMappingAddress` kernel API for the allocation of System-Page Table entries.

### 2.2.2 Section Object

They are managed by the operating system. Mapping of the section object is required before it can be accessed. Before the content of a section object can be accessed they should be mapped. When the virtual address range is allocated for the object and is accessible through address range, then the object is mapped.

Section object can be mapped to more than one place. It is one of the convenient tools to share the memory between them. Section objects are also called as memory mapped files. Section object can be classified into two parts

-- Page Backed file and File Backed. Page-backed section object is used for the temporary storage of data between two processes. The section is initially created empty then it can be mapped to any address space. File Backed object is attached to a file in the physical space. It will contain the contents of the file to which it is attached. Any changes or the modification which is made in the object will be reflected in the file. It provides more convenience to access a file using the object since, instead of using a cumbersome API, such as ReadFile and WriteFile, object can directly access the file in memory using a pointer. They are generally used for loading the executable image.

## 2.3 Virtual Address Descriptor

Virtual Address Descriptor (VAD) is used for managing the individual process allocation. It is a binary tree which contains every address range in use. There are two kinds of address ranges. Mapped allocation and private allocations. A mapped allocation comprises the memory mapped files such as executables and other files in the address space. Private allocation which are generally used for heap and stacks.

### 2.3.1 User Mode Address Space

In user mode allocations there can be different types of memory. These types include private allocations, Heaps, Stacks, Executables, and mapped view sections. In private allocation, application requests a block of memory using VirtualAlloc Win32 API. It has to be noted that it can allocate the whole pages. They are mainly used for allocating stacks and heaps. The functions such as malloc or system heap API such as HeapAlloc are being used for the allocation of the heap memory. Heap manages the memory such that the block of memory can be allocated and freed as required. VirtualAlloc API, can be used by an application to implement its own heap by directly allocating private block. A stack is allocated for each thread while it is being created. User mode thread is private allocations. System allocates a stack for every thread while it is being created. Executable code is loaded in memory as a memory mapped file.

### 2.3.2 Memory Management in windows

Set of windows32 API can be used to access the virtual memory manager. It can be used to directly allocate and free the memory in the user mode address space.

VirtualAlloc is used to allocate a private memory block in a user mode address space. The size of the block must be page aligned. The block of memory must not be variable. Block of memory can either be reserved or it can be committed. A reserved block differs from the committed block in the sense that it only reserves address space. VirtualProtect is used for to enforce protection settings. It defines if the memory block is readable, writeable or executable. VirtualQuery function provides the details such as type of block like (private, section or an image) and if the block is reserved, committed or unused. VirtualFree function frees the private allocation block. ReadProcessMemory and WriteProcessMemory are the two windows APIs that can be used to access another processes' memory space.

### 2.3.3 Objects and Handles

The various types of kernel objects are section, files, and device objects, synchronization objects, processes and threads. Centralized object manager component is used by the Windows kernel manager. Objects such as windows, menus and device context are managed by separate object manager which are implemented inside WIN32K.SYS. Kernel directly accesses the object using direct pointer to object data structure; however, applications uses handles for accessing individual objects. Each entry in the handle table comprises pointer to the underlying object. Besides the object pointer handle entry also contains access mask which determines the type of operations to be performed using the specific handle. Object access mask is a 32-bit integer, the upper 16 bits comprises the generic access flag such as GENERIC\_READ and GENERIC\_WRITE. The lower bit comprises object-specific flags such as PROCESS\_TERMINATE. This allows terminating a process using its handle. KEY\_ENUMERATE\_SUB\_KEYS this allows to enumerate subkey of an open registry key.

### 2.3.4 Named Objects

They are arranged in the hierarchical directory. Conventional Win32 named objects such as mutexes are stored in BaseNamedObject directory. The entire named object Win32 APIs uses this directory. All the device objects are under the Devices directory. The directory contains the entry for each device driver. It also comprises devices which are not connected to the system. It comprises logical devices such as TCP, and physical devices like Harddisk0. GLOBAL?? is the symbolic link directory. They are old-style name for kernel objects. Unnamed kernel objects are identified by their handles or kernel object pointer.

## 2.4 Processes and Threads

Process has its own memory space or basically it comprises of private set of basic run-time resources. For communication between the processes, IPC or inter process communication like pipes, and sockets are used. Threads are light-weight processes and exist within the process; however, like processes they require few resources. A process is an execution stream in the context of a particular process state. A thread is a single sequence stream within in a process. Execution stream is a sequence of instructions whereas process state comprises registers, stack, memory, open file tables and signal information. There is one process at a time in the case of uni programming. In the case of multiprogramming, there are multiple processes at a time. In multi programming, resources need to be shared between the processes. One of the critical resources is CPU, OS executes on one process, and then takes away CPU from the process and let another process executes it. It should ensure all the processes get their fair share of CPU. Process abstraction is performed by context switching to switch from one process to another. Details of context switching are discussed in the section 2.4.1. A thread is an execution stream in the context of a thread state. Multiple threads share the same address space. Multiple threads read and write to same memory. However, each thread has its own register and stack. Operating System will have its own thread for each distinct activity and the thread will perform operating system activity on the thread. A thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals. Threads like processes share CPU and there is only one active thread at a time. Threads within a process execute sequentially and can create children. If one thread is blocked then other thread can execute. However threads differ from the process in the respect that unlike processes threads are not independent of one another. Unlike process all threads can access every address in the task. Threads are designed to assist each other; however, processes might or might not assist one another since process may originate from a different user. Since thread can share common data, they do not need to use inter-process communication. Threads can take advantage of the multiprocessors. Threads only need stack and storage for registers hence they are cheap to create. Thread makes use of little resources of an operating system. They do not need new address space, global address space, program code or operating system resources. Since in the case of thread, only PC, SP and registers are stored, context switching is fast when working with the thread. User level threads are implemented in the user-level libraries and are implemented as if they are single-threaded processes. User level thread does not require modification to

operating system. Each thread is represented by a PC, register and a small control block which is stored in the user process address space. In the case of user level thread, switching between the threads and synchronization between the threads can be done without the intervention of the kernel. Switching of a thread is cheap as compared to the procedure call. Because of the lack of coordination between the thread and the operating system kernel, process as a whole gets only one time slice which is irrespective of whether process has one thread or 1000 threads. User-level threads require non-blocking system call, that is, a multithreaded kernel, otherwise entire process will be blocked in the kernel. If one thread causes a page fault, the process blocks. In this method the kernel knows about and manages the threads. Instead of thread table in each process, the kernel has thread table for all the threads in the system. Kernel makes use of process table to keep track of processes. Since the kernel has knowledge of the threads, it might happen that the scheduler may decide to give more time to one process having a large number of threads than process having a small number of threads. The main drawback of kernel level threads is they are slow and can be inefficient. Since kernel must manage and schedule threads as well as processes, it requires thread control block (TCB) for each thread to maintain information about threads. Hence there is significant overhead and increased in kernel complexity. Threads do not require space to share memory information, open file of I/O device in use hence they are much faster to switch between the threads. It is relatively easier for a context switch between the threads. Moreover, unlike processes threads allow sharing of information which cannot be shared in processes. This includes sharing of code section, data section and operating system resources such as open file, etc. If the kernel is single threaded, system call of one thread will block the whole process and the CPU may be idle during the blocking period. Multiprocesses have disadvantage over the threads since in thread it might happen that one thread might overwrite the stack of another thread. However, it also has to be noted that threads are meant to cooperate on a single task. Threads are useful for satisfying the requests for a number of computers on a LAN. Threads are suitable for applications which have more than one task at a time. Any sequential process which cannot be divided into parallel task will not benefit from thread. Code section, data section and operating system resources such as open file are shared with other resources. However, it is allocated its own stack, register set and a program counter. The creation of a new process, is different from the thread. All the shared resources of a thread are needed explicitly for each process. Hence the two processes will have a different copy of code in the main memory to be able to execute. This makes creation of new process costly as compared to new thread.

### 2.4.1 Context Switching

Context switch, also known as process switch, involves switching of the CPU from one process or thread to another. Process is an executing instance of a program. Threads are light weight process containing program counter and a stack. Contents of the CPU's registers and program counter at any point of time define the context. Contents of a CPU's register and program counter at any point in time define the context. Context switching involves suspending the progression of one process and storing the CPU's state for that process somewhere in memory. It then involves retrieving the context of next process from memory and restoring it in the CPU's register and returning to the location inside the program counter. It can be described as the kernel suspending the execution of one process on the CPU and resuming the execution of another process which has been suspended.

#### 2.4.1.1 Context Switches and Mode Switches

Kernel mode is a privileged mode of the CPU on which kernel executes and it provides access to all the memory locations and other system resources. Context switches can happen only in the kernel mode. Other applications can execute in user mode however they can execute portions of the kernel code via system calls. System call comprises request in a operating system by an active process for a task performed by the kernel. The task can be input/output, that is, any movement of information is to or from the combination of the CPU and main memory. Context switching is an essential feature for multitasking operating system. As discussed in context switching, multiple process execute on a single CPU without interfering each other. Thus, context switching also provides illusion of concurrency. Context switching of a process happens as a result of the scheduler making the switch when a process has used up its CPU time slice or it can be as a result of hardware interrupt. Hardware interrupt is a signal from the hardware such as keyboard, mouse, modem, or system clock to the kernel, than an event such as key press, mouse movement has occurred. Context switching can be done by using hardware or by software. Hardware context switching is supported in platforms like Intel 80386 and higher CPU. Software context switching can be done on any CPU rather than hardware context switching so as to obtain improved performance. In the case of hardware context switching all the CPU states are saved. In the case of software context switching only the required states are stored. Whereas in the case of hardware context switch all the CPU states are saved. Software context switching allows for the possibility of improving the switching code, thereby further enhancing efficiency, and that it permits better control over the validity of the data that is being loaded.

Cost of context switching can be from order of nanoseconds for each of the tens or hundred of switches per second. It can be one of the costly operations on an operating system.

To explaining this with an example, the **GetMessage** function retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval. **GetMessage** extracts the next event however there are many times there is no message. In such a scenario, **GetMessage** enters inside the waiting mode. It stays in the mode until the new input is available.

## 2.4.2 Synchronization Objects

Even though threads though provide flexibility, however synchronization of multiple threads is a challenging task. Threads will have to share the same data objects between them. So multithreaded applications require the proper design of a data structure and efficient locking mechanism. In a multithreaded environment, if two or more threads can be blocked or put in a special wait state by the kernel. They remain in the state until the wait condition is satisfied. Hence the synchronization objects are supported by the kernel. Scheduler has to be aware of the existence in order to determine when the state has been satisfied. Critical Section, mutex, Semaphore, Event and the metered sections are commonly used synchronization objects.

### 2.4.2.1 Critical Section

They are one of the most primitive synchronization objects in Win32. They are used for exclusive access to shared data between threads within a single process. The critical section code executed entirely in the user mode makes it very fast. There is no penalty on transition between user and kernel mode. It has to be noted that the events are kernel objects. So in the case of contention, the transition to kernel mode must be made. The transition time is not significant compared to the time the thread is blocked. Since the critical section does not have a named kernel object associated with it, its main disadvantage is it cannot synchronize access between processes.

### 2.4.2.2 Mutex

Mutex is a kernel object. It is implemented as a kernel object. It can synchronize between processes/threads; however, this ability comes at the stake of speed. Whenever process calls wait function such as

**WaitForSingleObject**, the transition between user mode and kernel mode is made. Mutexes can be used to synchronize the exclusive access. Only one thread can acquire mutex at a time. A thread can acquire mutex only on two conditions, either it has to wait till the thread having the mutex is released or until the thread holding the mutex terminates. In case of multi threading, it might happen that two or more threads are waiting for the mutex. In such a scenario, threads will receive the ownership of the mutex in the order in which it was received.

#### 2.4.2.3 Semaphore

They are similar to mutexes and are implemented as kernel objects. Hence they can work across the processes and are relatively slow. Semaphore, besides providing exclusive access to the shared object can be used for resource counting. Where mutexes and critical sections allow only one thread to gain access to a shared resource at a time, semaphores allow a set number of threads to gain access to a shared resource. It may happen that the maximum number is exceeded. In such a scenario, thread which requests the ownership of the semaphore will either enter a wait state or until another thread releases semaphore.

#### 2.4.2.4 Event

Events are primitive kernel synchronization objects on which other synchronization objects can be built. By themselves they are relatively slow, but they can synchronize access between processes by using named events. Depending on how they are used, events are capable of providing resource counting, but do not keep track of the count by themselves. Standard Win32 API `WaitForSingleObject` or `WaitForMultipleObject` is being used for waiting an event.

#### 2.4.2.5 Metered Section

Metered sections are an extension of critical sections. They provide ability to synchronize the thread across processes and they provide resource counting semantic similar to the semaphore kernel object. Metered section was to develop to achieve synchronization with the speed of a critical section and the cross-process resource counting of a semaphore. It was also designed to make them compatible with all Win32 platforms.

## 2.5 Process Initialization Sequence

The first step in the process initialization sequence is the creation of new process object and new address space. A new API is created for a process object and memory allocation is done when the Win32API `createprocesses` is called. `CreateProcess` maps NTDLL.DLL and the program executable into the newly created address space. `CreateProcess` not only creates process thread but also allocates address space. The first threads executes inside the `LdrpInitialization` function inside NTDLL.DLL. Primary executable import table is recursively traversed by the `LdrpInitialization`. It then performs the mapping of every executable which is required for executing the primary executable. The control is then passed to `LdrRunInitializeRoutines` which is internal NTDLL.DLL routine responsible for initialization all statically linked DLL which is presently loaded in the NTDLL.DLL. The initialization process consists of calling each DLL's entry point with the `DLL_PROCESS_ATTACH` constant. Once all the DLL are initialized, `LdrpInitialize` calls the thread initialization routine. This routine is `BaseProcessStart` function from `kernel32.dll`. This function in turn calls the executable `WinMain` entry point. Once the call the `WinMain` entry point is made, the initialization routine is complete.

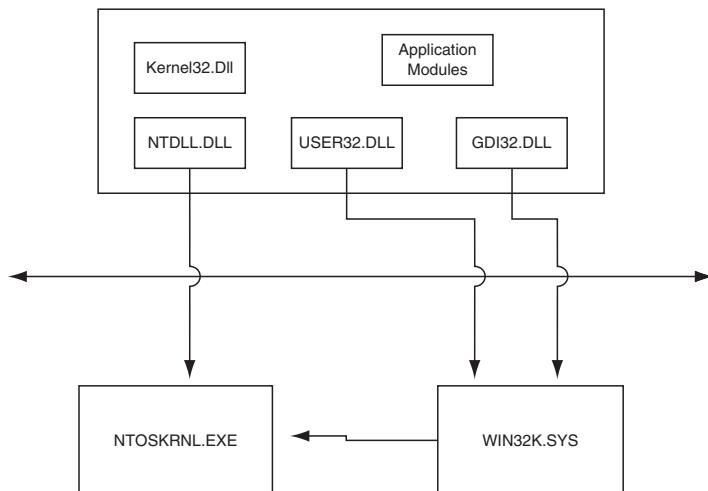


Figure 6.0 showing Win32 interface DLLs and their relation to the kernel components.

### 2.5.1 Application Programming Interface

An application programming interface (API) is set of functions which are used for interaction between the application and the operating system. The core Win32 API comprises 2000 APIs which can be divided into three parts. Kernel, USER and GDI. Figure 6.0 shows the relationship between the APIs. KERNEL32.DLL contains the Kernel API's. They include non-GUI related API which includes file I/O, memory management, object management, process and thread management. KERNEL32.DLL calls low level API from NTDLL.DLL. GDI32.DLL implements all the GDI APIs. They are implemented in the WIN32 kernel. Actual interface to the Windows kernel is Native API. It does not include any graphic related API. Set of functions exported from NTDLL.DLL and from NTOSKRNL.EXE comprises native API. It has to be noted that the Native API starts with Nt or Zw. In the user mode implementation both the APIs point to the same piece of code; however, it has to be noted that in the Kernel mode they are different. Nt version comprises the actual implementation of API whereas Zw are stubs that gets called through the system-call mechanism. Calling from system call mechanism, from kernel mode, ensures that the call is from kernel mode. Otherwise the call will be inferred to as a call from user mode. For user mode calls, it will be verified that the parameters will contain user mode address. Zw APIs simplify the process of calling function since in this case regular kernel mode pointers can be passed.

When user mode applications make a call to the kernel function system call mechanism takes place. The validation of parameter takes place at the usermode side of API, after which the parameters are passed to the kernel mode to execute the requested operation. The validation of parameters ensure that the invalid address is not called. Invalid address may result in kernel crash or it may result in taking control of the system. User mode code invokes CPU instructions. The CPU instructions instruct the processor to switch to privileged mode and make a call to dispatch routine. The dispatch routine makes calls the specific system function requested from the user.

In Windows 2000 and earlier system would invoke call to interrupt 2E for making a call to kernel. In a typical sequence of instruction first the EAX register is loaded with the service number followed by EDX register pointing to the first parameter to the kernel mode function. The instruction int 2e is invoked; processor uses IDT Interrupt Descriptor Table, to determine which interrupt handler to call. The IDT tells which routine to call whenever an interrupt or exception takes place. The interrupt 2E points to an internal NTOSKRNL function called as KiSystemService. KiSystemService is a kernel service dispatcher which verifies that the

service number and stack pointer are valid. `KiServiceTable` array comprises pointers to various kernel supported services. The request number loaded in the `EAX`, is used by the `KiSystemService` for indexing into the `KiServiceTable`. The `int 2e` stores the current value of `EIP` and `EFALGS`.

Current version of operating system uses different mechanism for performing this. Instead of invoking an interrupt to perform the operation, `SYSENTER` instruction is used to perform the operation. `SYSENTER` is a kernel mode switch instruction that calls predetermined function whose address is stored at special model specific register `MSR` called as `SYSENTER_EIP_MSR`. The contents of `MSR` can be accessed from kernel mode. `SYSENTER` does not store state information so by making a call to the `SystemCallStub` operating system records the state of current user mode stub in stack. This recorded user mode stub is used, when the kernel completes the call and needs to go back to the user mode.

## 2.6 Reversing Windows NT

The section discusses some of the basic techniques for reversing Windows NT. The `KERNEL32.DBG`, `NTDLL.DBG`, `NTOSKRNL.DBG` files will be required to debug kernel component. `USER32.DBG`, `GDI32.DBG`, `CSRSS.DBG`, `CSRSRV.DBG`, `WIN32K.DBG` are the `DBG` files that are required to explore `USER` and `GDI` component. Using symbolic loader these `DBG` files are converted into the `.NMS` files `stdcall` and `fastcall` are the two compiler calling conventions. Most of the functions in Windows NT follow either of these calling conventions. The file `NTPSKRNL.EXE` comprises many functions which follow `fastcall` calling convention. The parameters are pushed from right to left by the caller and the parameters pop off the stack by the called function. The `stdcall` calling convention provides the inherent advantage that the code is compact. This is because of the fact that the parameters reside in one place. It has to be noted that since fixed number of parameters pop off, this calling convention cannot support variable number of arguments. `cdecl` calling convention can be used to support this. The `fastcall` calling convention is similar to `stdcall`, with the difference being the first two parameters are passed in registers instead of being passed to a stack. Kernel data variables can be used to control the output of debug messages. These bits can be used to get more debug information from the operating system.

### **2.6.1 ExpEchoPoolCalls**

When the value of the variable is set to 1, information about each memory allocation/deallocation which is performed can be obtained by using the function.ExAllocatePoolWithTag and ExFreePool. The function provides the information including size of the region allocated, if the pool which is used is allocated or deallocated and the type of memory.

### **2.6.2 ObpShowAllocAndFree**

When the value of variable ObpShowAllocAndFree is set to 1, information about creation and destruction of each executive object can be obtained. The information also provides the type of object. Like if the object is Key, Semaphore and so on.

### **2.6.3 LpcpTraceMessages**

When the value of the variable LpcpTraceMessages is set to 1, information about local procedure call (LCP) function can be obtained.

### **2.6.4 MmDebug**

Different bits in the variable indicate different message generated by the memory management system.

### **2.6.5 NtGlobalFlag**

One bit of this variable enables the debug messages. Other bits control the validations performed by the operating system and general operation of the operating system. GFLAGS utility provides detailed description of individual bits of NtGlobalFlag. The value of this variable is inherited by a variable in NTDLL.DLL during the process startup. NTDLL.DLL uses the second bit of this variable to show the loading of a process. During process startup, NTDLL gets the value of this flag and sets its internal variable ShowSnap to 1 if the second bit is set. Once this bit is set, the behavior of the PE executable/DLL loader.c can be monitored. Windows NT will show names of all the imported

DLLs, plus it will show a real set of DLLs required to start an application. It will also show you the address of initialization functions of each of these DLLs as well as a lot of other information.

### **2.6.6 SepDumpSD**

When the value of the variable is set to 1, the security descriptor is dumped in the security handling related code.

### **2.6.7 CmLogLevel and CmLogSelect**

The variables provide control over the debug messages given by the registry handling code. The maximum value of CmLogLevel is 7. The volume of message generated by the operating system can be controlled by setting the individual bit in the CmLogSelect.

## **2.7 Security Features in Vista**

Vista provides various security features. The following section discusses the details of these security features.

### **2.7.1 Address Space Layout Randomization (ASLR)**

ASLR involves randomization of starting point of memory in stack and heap. It makes it difficult for an exploit to locate the address of system API. Since it becomes tough to locate the address of API, it becomes tough to run an arbitrary code. In the case of other operating system, such as Windows XP, the starting address of system API is known to attacker. Even though the starting address may differ depending on the service pack level of the system, however it can easily be calculated. ASLR includes randomization of address of images and DLL, starting address of each stack and starting address of each heap allocation.

One of the common attacks is to force an application to load the DLL. An attacker can write a path into buffer with known location and redirect execution to place where eliminating the precondition needed by the attacker. The attacker has to know the address where it should be jumped. ASLR is

done once per reboot. DLL will be loaded once per reboot. If all the processes using a particular DLL unload ASLR, it would be loaded in the random place in the next load. Some network service restarts itself on failure. This gives attacker a chance to find where to call system API. Hence it is recommended that the services be configured to restart automatically a small number of times. ASLR provides protection against the attacks of worms. However, if an application has format string vulnerability or information disclosure vulnerability, it might be possible for an attacker to learn the memory locations needed to over come this mitigation.

The randomization is in the second most significant byte of the address. To reduce virtual address space fragmentation, the library is relocated across 256 different possible addresses.

When the two DLLs are loaded in the overlapping ranges, then the last DLL which has to be loaded is relocated to a different address. The relocation process can be time consuming since it will involve changing every fixed address in the entire DLL to reflect the new starting point. Since relocation is an expensive process, hence relocation should be prevented. ASLR implementation deals with the performance concerns. It delays fixups until that page of the DLL is loaded into memory. Generally, console applications will use only a dozen or so functions exported by Kernel32.dll and hence would require fixing up the pages required to load those functions.

DLLs do not set their own address space, Vista packs them in with as little slack space between DLLs as possible. Since all the DLLs are loaded in the contiguous space, there is effectively more space for other applications. It also increases the cache performance.

### 2.7.2 Stack Randomization

Under the protection mechanism enforced by stack randomization, the base address for every thread is changed. This makes it difficult for an attacker to find a place to jump to within an application. */dynamicbase* in the linker options has to be used to get stack randomization. Even though the starting address of the executable command is randomized, the offset between the various code elements remains constant. Address of global variables is randomized as well in the case of stack randomization. Generally it is not recommended to store function pointer in global variables, stack randomization makes it difficult to attack Encoded pointers. In a multithreaded application, address of the stack buffer has been unpredictable in multithreaded application. Even though the stack buffer is unpredictable in

multithreaded application, the location of stack for the main thread is randomized. The offset between the stack and the main module code isn't fixed from one instance of the application to the next. Offset between the stack and the main module's code is altered from one instance of the application to the next. There is no effect of stack randomization on performance or compatibility issues.

### 2.7.3 Heap Defenses

The option /GS has made stack overflow difficult to exploit, however heap overflow is simpler to exploit. As adversary can make execution flow to jump into some spot in heap, it cannot control the precise address location. One of attacks can be to put a series of NOP sleds followed by the shell code. This will result in execution of NOP followed by the shell code. It might not be possible to put a large amount of data in the heap so an alternative technique, Heap Spraying can be used. The technique involves large copies of shell code in the heap. The execution will jump to shell code, since a large amount of allocation would end up in another location.

In a double free vulnerability, a pointer is accidentally freed twice. When the chunk is freed, it is added to the free list for future use. It can be later allocated and used, before getting freed again. Attacker can arbitrary set the forward/backward links in a heap chunk. On the same free list, a free chunk is added to a doubly linked list of other chunk. These forward and backward pointers are stored within the chunk data itself, that is, at offset 0 and 4 of the chunk data. An attacker has control over the FreeList.Flink and FreeList.Blink in the double freed chunk, making 4-byte overwrite trivial.

```
char* ptr_1 = new char[16];
.....
delete[] ptr_1;
// New allocation of some more memory
char* ptr_2 = new char[16];
// Note that ptr_1 has the same length as ptr_2
delete[] ptr_1;
// This will free ptr_2!
// Further memory allocation
char* ptr_3 = new char[16];
// ptr_3 will now be used to write memory that
// the code dealing with ptr_2 thinks is validated
```

Figure 7.0 C code showing double free pointer vulnerability

As shown in the code in figure 7.0, it can be seen for a double free pointer vulnerability, there is a pattern of *alloc(1), free(1), alloc(2), free(1), alloc(3)*. The pointers 1,2 & 3 all point to the same address. The efficient heap behavior reallocates recently freed memory which is of the same size. The function which requested *alloc(2)*, has a pointer to memory controlled by the function that called *alloc(3)*. To exploit, attacker has to control the memory written in to *alloc(3)*. Another attack pattern will have the *alloc(1), free(1), alloc(2), use(1), free(1)* sequence of instructions.

```
char* ptr_1 = new Cfoo;
// code comes here .....
delete[] ptr_1;
// Another allocation the same size follows it
// Note that ptr_1 and ptr_2 point to the same memory
char* ptr_2 = new Cfoo;
// Copy some data into ptr_2
// code will change data at ptr_1. It will not know
// that ptr_2 has changed things
// If ptr_1 is a class, destructor will be called.
delete[] ptr_1;
```

Figure 8.0 C code showing double free pointer Vulnerability

As shown in the Figure 8.0, in this case, the code using *ptr\_2* is changing the contents of the buffer pointed to by *ptr\_1*. It has to be noted that the *ptr\_1* contains a valid data. There is some potential for the usage of *ptr\_2* to attack *ptr\_1* and in this case, the converse is true as well—the usage of *ptr\_1* could very easily cause the data kept in *ptr\_2* to become invalid.

To prevent such a kind of exploit, pointers are set to null, when they are freed. However, the method of setting pointer to null will not be of much use if there are multiple copies of the same pointer. If the pointers are set to null then the *use(1)* will result in null deference crash. By setting pointer to null, the pattern *alloc(1), free(1), alloc(2), free(1), alloc(3)*, will be non exploitable. This is because functions 2 and 3 will have allocation in different space.

These conditions can be located by the debugging assert which will help to locate and fix these conditions. At run time the second delete will begin. Smart pointer classes are the other technique which can be used to fix all the double-free bugs.

The effect of heap overrun is dependent upon the heap manager which is being used. In the case of Windows, heap places control data before and after allocation, hence attacker can target both the control and the heap data which

is kept on the heap in adjacent memory location. In Windows Vista there have been several improvements in the heap.

Vista performs a check for the validity of forward and backward links. Free block has the address of previous and next free block. These addresses are stored immediately after the block header. The value of the forward link is the value to write and value of the backward link is where to write the forward link value. This will result in arbitrary 4 bytes being written anywhere in memory. Modification ensures that the structure at those locations properly point to where it started. This was delivered in Windows XP SP.

The block header is XORed with a random number. This makes determining the value which needs to be overwritten very difficult. The performance impact is small; however the benefits are very large. The previous 8-bit cookie has been repurposed to validate a large part of header. As discussed earlier, the heap base is randomized and the function pointers which are used by heap are encoded. Vista provides termination on heap corruption in an application. However, it might happen that the exploit happens before the heap manager notices corruption. In the earlier version, the default behavior when the application heap became corrupted was to leak the corrupted memory and keep on executing.

Low fragmentation heap (LFH) is generally used when program allocates large amount of memory in various allocation sizes. LFH allocates blocks of memory which are as long as 16 kilobytes (kb). For memory block which are larger than 16KB, the LFH uses the standard heap. Fragmentation is minimized by the LFP algorithm and improves Win32 heap allocation performance. In comparison to Windows heap, the LFH are more resistant to attacks. Vista makes use of LFH.

## 2.7.4 NX

NX, stands for short for “No eXecute”. As per the NX, if a page of memory, whether it is on stack or heap is writeable, should not execute code from that page. When a DLL is loaded after process initialization, the operating system has to allocate pages and write instructions into process memory which a system should be able to execute. If a shell code could first cause *VirtualProtect* to be called with correct parameters, NX is then defeated. *NtSetInformationProcess* disables NX for an entire process, unless the application has been compiled with */NXCOMPAT*. This functionality allows

for backward compatibility and allows an application to continue to work if it happens to load a DLL that isn't compatible with NX protection. Combination of NX and ASLR can stop most of the attacks. In many of the system the default BIOS option sets the NX to off.

Similar to ASLR and the heap settings, NXCOMPAT flag is set process wide. NXCOMPACT flag is set in the linker option then the application will be running NX irrespective of the option set in Windows. If NXCOMPAT:NO is set, then NX will not apply to the application. NX option does not pose any performance penalty or raises any performance impact. However, there can be compatibility problem when there is an exception. If requirement of assembler can be predicted, then it is advisable to write to memory and disable write at the same time when execute is enabled on the page. In case applications permit plugins, the plugins must be removed out of the processes.

### 2.7.5 /GS

/GS option places a randomly generated cookie placed between the return address and the local variable on the stack. The cookie will guard the EBP register which was pushed on to the stack. /GS is effective in preventing off-by-one overflow attacks.

There will be several structured exception handlers SEH in windows. The `_try` keyword declares a block that has an exception handler. The `_except` keyword declares a block which behaves similar to a block declared with `catch` in C++. When exception is raised, the exception handler is raised, exception record is searched to determine if the exception needs to be handled. The program might continue execution after the handler, fixes the problem and resumes the execution after the handler fixes the problem and resumes execution. A `_finally` block is a method for a C program to behave very similarly to how a C++ application would use a destructor. The block of code inside `_finally` is guaranteed to gets executed

```
_try
{
    // Code come here
}
_except( EXCEPTION_EXECUTE_HANDLER )
{
    // Code gets executed when there is some error.
}
```

*EXCEPTION\_REGISTRATION* structure is pushed onto the stack whenever an exception handler is registered. The structure *EXCEPTION\_REGISTRATION* contains a pointer to the next *EXCEPTION\_REGISTRATION* structure along with the pointer to the next *EXCEPTION\_REGISTRATION* structure and the address of the current exception handler. It has to be noted that there is a function pointer on the stack which can be overwritten. To exploit this, buffer has to be overwritten with the address of the attacker choice. This attack will work, regardless of the code internal function. However, the overwrite should extend far enough to hit the exception handler or an arbitrary DWORD overwrite and condition for an exception should be caused prior to the function exiting normally. The Visual Studio 2005 compiler treats calling an exception as if the function has exited normally and checks the security cookie prior to executing the handle.

For a 64-bit code, the exception records are compiled into the binary. They are not kept onto the stack. Hence 64-bit executables are much safer at least from the SHE attacks.

## 2.7.6 Pointer Encoding

Windows Vista supports pointer encoding, which provides developers with an ability which makes it harder for an attacker to overwrite a pointer with a valid value. This enables them to prevent them from buffer overrun. C and C++ provide pointer which points to arbitrary memory locations. Code can read from and write to arbitrary memory locations.

```
class foo {
public:
    foo() {
        dest = new char[64];
        data = new char[10];
    }
    ~foo() {
        delete [] dest;
        delete [] data;
    }
    const char *Write_Data(const char *src, *src1) {
        if (dest) strcpy(dest,src);
        if (data) strcpy(data,src1);
        return src;
    }
private:
    char *dest, *data;
};
```

Figure 9.0 sample C code without pointer encoding

In the code shown in Figure 9.0, adversary has control over the src. When the function Write\_Data is called, overlong value is passed to the src, it will overwrite data pointer. The pointer data will be few bytes higher in the memory. The code shown in figure 10.0 the copies the src1 to data pointer and effectively can write in memory This is heap overrun vulnerability. .

```

.
class foo {
public:
    foo() {
        dest = (char*)EncodePointer(new char[64]);
        data = (char*)EncodePointer(new char[10]);
    }
    ~foo() {
        delete [] DecodePointer(dest);
        delete [] DecodePointer(data);
        data = dest = NULL;
    }
    const char *Write_Data(const char *src, char src1) {
        char *dec_dest = (char*)DecodePointer(dest);
        if (dec_dest) strcpy(dec_dest,src);
        char *dec_data = (char*)DecodePointer(data);
        if (dec_data) strcpy(dec_data,src);
        return src;
    }
private:
    char *dec_dest, *dec_data;
};
```

Figure 10.0 shows the similar contrived C++ class code using pointer encoding

The code shown in Figure 10 is similar to the original code shown in Figure 9, except that the pointers are deemed long-lived and are encoded as soon as they are created and then decoded prior to use. The class destructor sets the pointers to *NULL* after the memory is deleted.

Hence, the following steps need to be followed for pointer encoding. First, memory is allocated or initialized and assigned the pointer to the address. Then the pointer is encoded. When the pointer is used, it is decoded to a temporary variable. When the pointer is not required, it is decoded and is set to free and *NULL*. If the pointer is overwritten, *DecodePointer* it will just give back a bad pointer.

### 2.7.7 Cryptographic API in Windows Vista

Windows Vista provides a new cryptography API called CNG:API (Cryptography Next Generation) which serves as a replacement for the old Cryptographic APIs. Figure 12.0 shows the design architecture of CNG. Not only CNG provides support for all algorithm for Cryptographic API, it also includes new algorithms. CNG provide two set of functions NCrypt\* BCrypt\*.

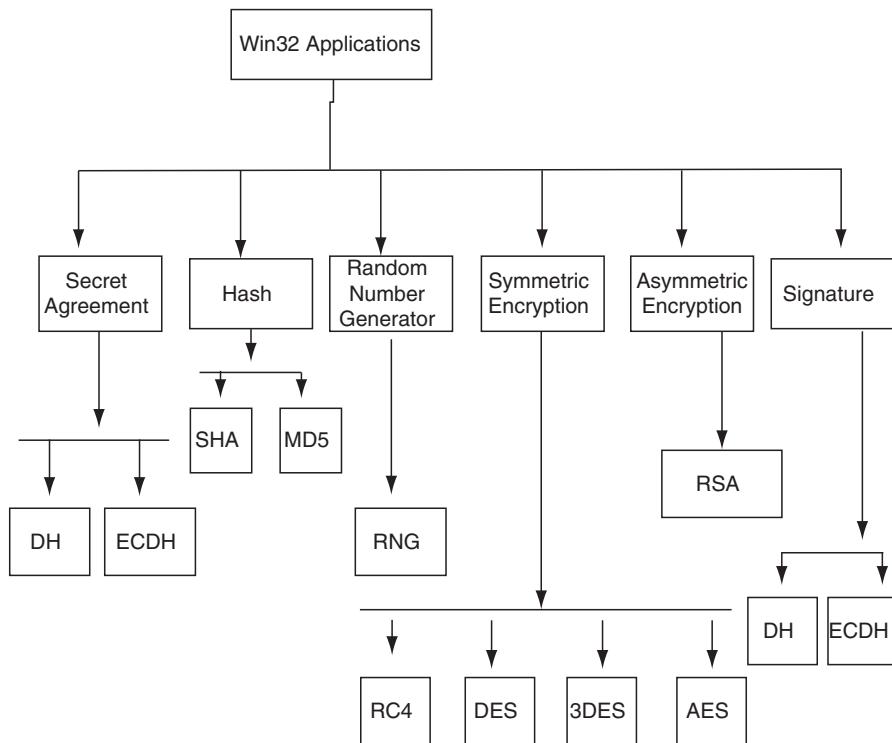


Figure 11.0 Figure showing Algorithms for CNG

**NCrypt\*** function: The function is a subset of CNG, which deals with key management, key persistence and key isolation and public key operations. NCrypt are available only to the user mode applications. NCrypt is the name of the DLL and the header file, which provides high-level key storage facility.

**BCrypt\*** function: The function is a subset of CNG, which provides low-level cryptographic primitives, which run in process with the applications. The keys are not stored, they are ephemeral. The keys are available in the kernel mode and provide cryptographic framework for both the user mode and the

kernel mode applications. BCrypt is the name of the DLL and the header file, which provides base services for CNG.

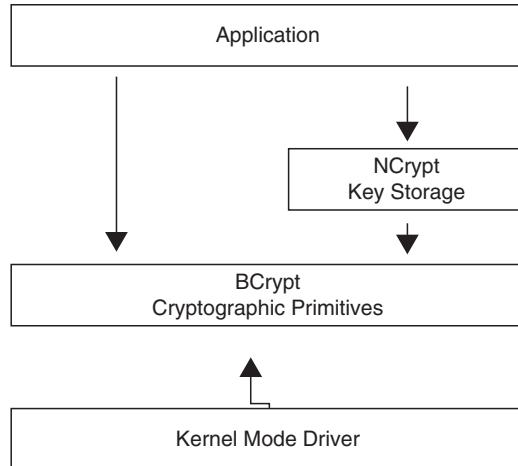


Figure 12.0 Showing CNG Architecture

The CNG API is built on logical cryptographic interfaces. It takes interface-centric approach which is different from algorithm-centric approach followed by most of the cryptographic algorithms. It provides inherent advantage in terms of flexibility for an application developer to replace an algorithm use by an application which is found to be flawed.

BCRYPT\_HANDLE is used for identifying the CNG objects which are defined by the BCrypt. Algorithm is loaded by the BCryptOpenAlgorithmProvider function, which loads an algorithm provider based upon the choice of an algorithm and returns a handle for use in subsequent calls to a CNG function.

### 2.7.8 Crypto-Agility

Increase in processor speed and developments in algorithms make cryptographic algorithms agile. Increase in speed of processor makes it feasible for an algorithm to be cracked in a reasonable time. Some of the hash functions such as MD4, MD5, SHA-1 are considered to be insecure.

### 2.7.9 Crypto-Agility in CNG

Cryptographic constants are strings rather than numeric constants in CNG. All the cryptographic algorithms are predefined in *wincrypt.h* which makes it difficult to extend the cryptographic functionality as per the application needs. In CNG, adding an algorithm is possible. String constant can be defined for an algorithm. When an application uses the algorithm, CNG will load the crypto-provider which is registered to the name. Custom cipher-suites for SSL and TLS can be plugged in. *BCryptAddContextFunctionProvider*: can be used to add new plugins.

CNG does not require Microsoft to sign the implementation. Cryptographic provider can be created by the cryptographer. Also it is possible for an application to query CNG for supported algorithms. Figure 13.0 shows the algorithms supported by the default CNG provider in Windows Vista.

<b>Algorithm</b>	<b>#define</b>	<b>Standard</b>
RC2	<b>BCRYPT_RC2_ALGORITHM</b>	RFC2288
RC4	<b>BCRYPT_RC4_ALGORITHM</b>	
AES	<b>BCRYPT_AES_ALGORITHM</b>	FIPS 197
DES	<b>BCRYPT DES ALGORITHM</b>	FIPS 46-3, FIPS 81
DESX	<b>BCRYPT_3DES_ALGORITHM</b>	
3DES	<b>BCRYPT_DESX_ALGORITHM</b>	FIPS 46-3, FIPS 81, SP800-38A
3DES-112	<b>BCRYPT_3DES_112_ALGORITHM</b>	FIPS 46-3, FIPS 81, SP800-38A

Algorithm	#define	Standard
Elliptic Curve Digital Signature  Algorithm with Prime-256 curve	BCRYPT_ECDSA_P256_ALGORITHM	FIPS 186- 2, X9.62
Elliptic Curve Digital Signature  Algorithm with Prime-384 curve	BCRYPT_ECDSA_P384_ALGORITHM	FIPS 186- 2, X9.62
Elliptic Curve Digital Signature  Algorithm with Prime-521 curve	BCRYPT_ECDSA_P521_ALGORITHM	FIPS 186- 2, X9.62
Elliptic Curve Diffie-Hellman  Algorithm with Prime-256 curve.	BCRYPT_ECDH_P256_ALGORITHM	SP800- 56A
Elliptic Curve Diffie-Hellman  Algorithm with Prime-384 curve.	BCRYPT_ECDH_P384_ALGORITHM	SP800- 56A

Algorithm	#define	Standard
RC2	BCRYPT_RC2_ALGORITHM	RFC2288
RC4	BCRYPT_RC4_ALGORITHM	
AES	BCRYPT_AES_ALGORITHM	FIPS 197
DES	BCRYPT_DES_ALGORITHM	FIPS 46-3, FIPS 81
DESX	BCRYPT_3DES_ALGORITHM	
3DES	BCRYPT_DESX_ALGORITHM	FIPS 46-3, FIPS 81, SP800-38A
3DES-112	BCRYPT_3DES_112_ALGORITHM	FIPS 46-3, FIPS 81, SP800-38A

Figure 13.0 Algorithm supported by the default CNG.

CNG also supports two kinds of random number generators (RNG), and both are allowed under SDL: *BCRYPT\_RNG\_ALGORITHM* and *BCRYPT\_RNG\_FIPS186\_DSA\_ALGORITHM* are the two random number generators which are supported by CNG. However, it has to be noted that the CNG password-based key derivation function is missing from CNG.

### 2.7.10 Algorithm Providers

All CNG objects defined by BCrypt are identified by a *BCRYPT\_HANDLE*, and used to identify the CNG objects defined by the BCrypt. Initially the algorithm provider is loaded based upon the choice of algorithm and optional implementation. The function

*BCryptOpenAlgorithmProvider* is used to achieve the objective and the functions then return a handle. The handle is used in subsequent calls to CNG function. Error is indicated by the *NTSTATUS* type from the Windows Driver Kit. This is used both for user mode and kernel mode programs.

```
BCRYPT_HANDLE algoProvider = 0;

NTSTATUS status = ::BCryptOpenAlgorithmProvider(
    &algoProvider, algoName,
    implementation, flag);

if (NT_SUCCESS(status))
{
    // Code for algorithm provider comes here
}
```

Figure 14.0 showing the sample implementation of BCryptOpenAlgorithm

Generally value 0 is passed both for the implementation and flag parameters. The value 0 indicates that the default algorithm provider should be loaded for the particular algorithm identified by the algorithm name parameter. The NT\_SUCCESS macro shown in Figure 14.0 indicates if the value represents success or failure. It has to be noted that the loading of the algorithm can be an expensive operation. Hence once the algorithm is loaded, it should be re-used as much as possible. As shown in the Figure 15.0 algorithm provided can be unloaded by passing the handle returned by BCryptOpenAlgorithmProvider to the BCryptCloseAlgorithmProvider function. Zero must be passed for the flags parameters.

```
status = ::BCryptCloseAlgorithmProvider(
    algoProvider, flags);
```

Figure 15.0 showing the closing of the algorithm

### 2.7.11 Random Number Generation

BCryptGenRandom function is used for to generate random number. It fills in buffer with random generated value. BCRYPT\_RNG\_ALGORITHM algorithm identifier denotes the default random number identifier. The BCRYPT\_RNG\_FIPS186\_DSA\_ALGORITHM algorithm identifier is used to meet the Federal Information Processing Standards (FIPS). It can be seen that the function BCryptRandom function expects a pointer to UCHAR which identifies the buffer. The function BCryptGenRandom provides an optional flag which allows to provide entropy for random number generation algorithm.

```

BCRYPT_HANDLE algoProvider = 0;
NT_VERIFY(::BCryptOpenAlgorithmProvider  (  &
algoprovider, BCRYPT_RNG_ALGORITHM,0,0));
Int n= 20;
for(int a=0; a< n; ++a)
{UINT rand =0;

    NT_VERIFY(::BCRYPTGenRandom(
        algorithmprovider,
reinterpret_cast<PUCHAR>(&rand), sizeof(UINT), 0));
    Count <<rand << endl;
}

```

Figure 16.0 showing code for random number generator

When the flag BCRYPT\_RNG\_USE\_ENTROPY\_IN\_BUFFER is used then the value passed in the buffer is used as an additional entropy in calculating the random number and is returned in the same buffer.

### 2.7.12 Hash Functions

The algorithm provider and the hash functions are represented by objects in CNG. The functions BCryptSetProperty can be used to set the named properties in the named objects and the function BCryptGetProperty is being used to query the function name. Similar function is provided for handling object property with NCrypt. A new hash object is created by using the BCryptCreateHash function, which needs a buffer that is required for processing. In kernel mode, care has to be taken while allocating memory. Kernel mode also requires, managing the handle and hash table resources for the hash object.

First parameter of the function BCryptGetProperty's indicates the object to query whereas the second parameter indicates the name of the property. Third and Fourth parameters indicate the destination buffer where the property value is stored and the size of the buffer. The noofbyteCopied parameter is useful in the case where the size of buffer is unknown. Chunk of memory is committed to the buffer for the hash object. Only in the kernel mode it matters where the memory is stored. The hash object can be created by the bCryptCreateHash function

```
BCRYPT_HANDLE algoProvider = 0;

NT_VERIFY(::BCryptOpenAlgorithmProvider(
    &algoProvider,
    BCRYPT_SHA256_ALGORITHM,
    0,
    0));

ULONG hashBufSize = 0;
ULONG noofbytesCopied = 0;

NT_VERIFY(::BCryptGetProperty(
    algoProvider,
    BCRYPT_OBJECT_LENGTH,
    reinterpret_cast<PUCHAR>(&hashBufferSize),
    sizeof(ULONG),
    &noofbytesCopied, 0));
```

Figure 17.0 showing code for hash function

. The first parameter to the hash function indicates the algorithm provider which implements the hash interface. The second parameter indicates the handle to the hash object. The last two parameters indicate the hash buffer and its size. The hash object is destroyed by using BCryptDestroyHash function and then the function frees the hash object buffer. For flags parameters, zero is passed.

It is also possible to duplicate the hash object which is useful in the case when two or more hash values need to be produced based on some common data. The function BCryptDuplicateHash is being used to duplicate the hash function. The function will require a handle to the hash object for duplication along with the new buffer which it will use for processing. For duplicating a hash function first a single has object needs to be defined then it has to be duplicated one or more times. After the duplication has been achieved, the two hash objects contain the same state; however, they have no connection to one another. Unique data can be added to each one producing a hash value. If one hash object is destroyed it will not affect the other.

### 2.7.13 Symmetric Encryption

The function BCryptGenerateSymmetricKey function is used to generate symmetric key. It involves first creating an algorithm provider followed by determining the size of the object buffer. Once the size of buffer is determined, then the buffer of that size is allocated.

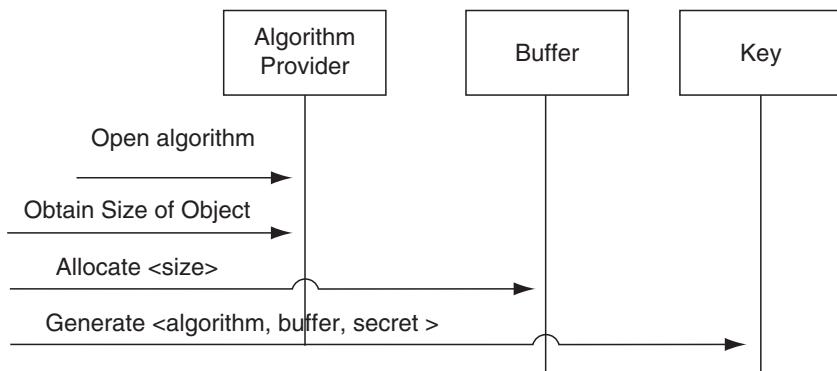


Figure 18.0. Creation of a Symmetric Key Object

The first parameter to the function BCryptGenerateSymmetricKey denotes that the algorithm provider implements a symmetric encryption algorithm. Implementation of symmetric key algorithm is denoted by the second parameter. Key buffer and its size are denoted by the second parameter. The buffer containing the secret key shared by the sender and receiver is denoted by the next two parameters. It can be any byte array or can be empty. Generally it is a hash of a password. Since the flags for this function are not defined, zero is passed as the parameter for the flag. BCryptDestroyKey function is used to destroy the key object. After the key object is destroyed, it is freed. The function BCryptEncrypt and BCryptDecrypt are being used for the encryption and decryption of data. The function is used both for the symmetric key and asymmetric key. The sender and the receiver share common properties. They require a key created with the same secret and with the matching properties values. They also require initialization vector which are equal. For symmetric encryption algorithm, size of data block for the algorithm needs to be determined. The size of block indicates the size of initialization vector. Block cipher encrypts a fixed sized block of plaintext into a block of cipher text of same size. Once the message to encrypt along with the initialization vector is prepared, BCryptEncrypt function can be used to encrypt the plain text message. BCryptEncrypt function is used for encryption. The first parameter for the function denotes the key to use for encryption. The next two parameters indicate the message which has to be encrypted. The fourth parameter provides additional padding information for asymmetric key algorithm. The flag BCRYPT\_BLOCK\_PADDING is used with symmetric algorithms. The function BCryptEncrypt is called again which contains buffer to receive the cipher text. The BCryptDecrypt function works in the similar manner. First the size of cipher text is determined, followed by

which the BCryptDecrypt function is called with the initialization vector and the plaintext buffer to obtain the resulting decrypted message.

### 2.7.14 Asymmetric Encryption

The advantage of Asymmetric algorithm is that the public key can be shared with anyone; however, the computational cost of the algorithm is higher than the symmetric key algorithm. Figure 18.0 shows the process for the establishment of the asymmetric key.

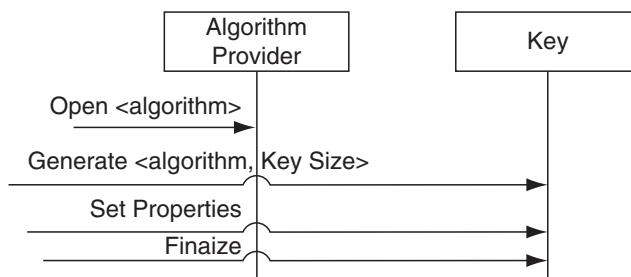


Figure 18.0 showing the creation of asymmetric key

The function BCryptGenerateKeyPair is used to create the public and private key. The first parameter to the function provides the details of the algorithm provider which has implemented the asymmetric key algorithm. Handle to the key object is received in the second object. Key size is indicated in the third parameter. The size of key is indicated in bits. The size of key affects the performance of the algorithm. Key size can also be used to determine the block size. The block size can be determined by dividing the key size by 8. BCryptSetProperty function is used to set the algorithm specific key properties. The properties are set after the key pair is generated. The BCryptFinalizeKeyPair function is being used to finalize the creation of key object. This is done before the key pair can be used. BCryptEncrypt and the BCryptDecrypt function can be used to encrypt and decrypt the block of data. It might happen that the message to be encrypted is provided in a buffer which is a multiple of the block size. The flag BCRYPT\_PAD\_NONE is used to denote that there is no padding. The flag BCRYPT\_PAD\_PKCS1 flag tells the algorithm provider to pad the input buffer to a multiple of block size. This is done by using a random number which is based on the PKCS-1 standard. The functions BCryptExportKey and BCryptImportKeyPair are used to export and import keys. To import symmetric keys BCryptExport function can be used to export symmetric key as well.

### 2.7.15 Signatures and Verification

Asymmetric key algorithms are used to create digital signatures. Public keys are used to generate signatures and private keys to verify the signatures. BCryptSignHash function is used to calculate the signature. The calculation of signature makes use of both the hash values as well as the private key for a digital signature algorithm. The generation of signature comprises two parts. The first part involves the calculation of the size of the resulting signature. The second part involves computation of signature. Additional padding information may also be required. The hash is computed independently and then the hash value, public key of the signature and the signature received to the BCryptVerifySignature function for verification. The function BCryptVerifySignature, returns STATUS\_SUCCESS in case of match between the signature and the hash value. STATUS\_INVALID\_SIGNATURE is returned if there is a mismatch in signature.

## 2.8 Conclusion

The chapter provides basics of the operating system, which is important for reverse engineering. Understanding the basic API offered by the operating system is useful in deciphering the programs. Virtual memory provides solution for memory management. Light-weight processes are called as threads. Threads perform context switching which is also termed as process switch. It involves switching of the CPU from one process or thread to another. Even though threads provide flexibility, synchronization of multiple threads is a challenging task. Critical Section, Mutex , Semaphore, Events, and Metered Sections can be used for synchronization of threads. In Windows NT KERNEL32.DBG, NTDLL.DBG, NTOSKRNL.DBG files will be required to debug kernel component. USER32.DBG, GDI32.DBG, CSRSS.DBG, CSRSRV.DBG, WIN32K.DBG are the DBG files that are required to explore USER and GDI component. Vista is the latest operating system. It provides address space layout randomization which makes it difficult to execute on a local address space. The code should be compiled with the /GS option and link with the /NXCOMPACT, /SAFESEH and /DYNAMICBASE option. Pointer encoding is supported in Vista, it makes it difficult for an attacker to overwrite a pointer with a valid value. Vista provides, CNG (Cryptographic Next Generation ) which provides replacement for the old cryptographic API. NCrypt and BCrypt are the subsets of CNG, which provide low-level cryptographic primitives.

# Portable Executable File Format

## 3.0 Introduction

PE stands for ‘portable executable’ file format. As the name suggests, the format can be portable across all the 32-bit operating system and can be executed on any version of windows. The format is also being used by 32-bit dlls and Windows NT device drivers. The WINNT.H header file defines the structure definition representation for the PE file format.

Understanding of PE file format is not only required for reverse engineering, but it is also required for understanding the concepts of operating system.

## 3.1 PE file format

PE stands for Portable Executable file format. It is generated using the Microsoft linker that has a .text section containing the code bytes concatenated from all the object files.

73D90000	00001000	CRTDLL	.text	PE header	Imag	R	RwE
73D91000	00010000	CRTDLL	.data	code, import	Imag	R	RwE
73D92000	00006000	CRTDLL	.data	data	Imag	R	RwE
73D93000	00001000	CRTDLL	.rsrc	resources	Imag	R	RwE
73D94000	00002000	CRTDLL	.reloc	relocations	Imag	R	RwE
77D00000	00001000	ADUAPI32		PE header	Imag	R	RwE
77D01000	00075000	ADUAPI32	.text	code, import	Imag	R	RwE
77E46000	00005000	ADUAPI32	.data	data	Imag	R	RwE
77E48000	0001B000	ADUAPI32	.rsrc	resources	Imag	R	RwE
77E56000	00005000	ADUAPI32	.reloc	relocations	Imag	R	RwE
77E70000	00001000	RPCRT4		PE header	Imag	R	RwE
77E71000	00003000	RPCRT4	.text	code, import	Imag	R	RwE
77EF4000	00007000	RPCRT4	.orpc	code	Imag	R	RwE
77EFB000	00001000	RPCRT4	.data	data	Imag	R	RwE
77EFC000	00001000	RPCRT4	.rsrc	resources	Imag	R	RwE
77EFD000	00005000	RPCRT4	.reloc	relocations	Imag	R	RwE
77F10000	00001000	GDI32		PE header	Imag	R	RwE
77F11000	00042000	GDI32	.text	code, import	Imag	R	RwE
77F53000	00001000	GDI32	.data	data	Imag	R	RwE
77F54000	00001000	GDI32	.rsrc	resources	Imag	R	RwE
77F55000	00002000	GDI32	.reloc	relocations	Imag	R	RwE
77FE0000	00001000	Secur32		PE header	Imag	R	RwE
77FE1000	00000000	Secur32	.text	code, import	Imag	R	RwE
77FE2000	00001000	Secur32	.data	data	Imag	R	RwE
77FE3000	00001000	Secur32	.rsrc	resources	Imag	R	RwE
77FF9000	00001000	Secur32	.reloc	relocations	Imag	R	RwE
7C800000	00001000	kernel32		PE header	Imag	R	RwE
7C801000	00003000	kernel32	.text	code, import	Imag	R	RwE
7C884000	00005000	kernel32	.data	data	Imag	R	RwE
7C889000	000066000	kernel32	.rsrc	resources	Imag	R	RwE

Figure 1.0 Showing .text, .data, .rsrc, .reloc

As shown in the figure 1.0 the .data contains all the initialized global and static data which can be classified into different categories. Initialized global and static data are classified under the .data section while .bss section contains

the uninitialized data. The .rdata section contains read only data such as string literals and constants, debug directory, thread local storage directory. The .edata section contains information about the functions exported from a DLL, while the .idata contains information about the functions imported by an executable or a DLL. Menu and dialog boxes are stored in the .rsrc section and .reloc section stores the details for relocating the image while loading.

40 5A	ASCII "MZ"	DOS EXE Signature
9000	DW 0090	DOS_PartPag = 90 (144.)
0300	DW 0003	DOS_PageCnt = 3
0000	DW 0000	DOS_ReloCnt = 0
0400	DW 0004	DOS_HdrSize = 4
0000	DW 0000	DOS_MinMem = 0
FFFF	DW FFFF	DOS_MaxMem = FFFF (65535.)
0000	DW 0000	DOS_ReloSS = 0
B800	DW 00B8	DOS_ExeSP = B8
0000	DW 0000	DOS_ChkSum = 0
0000	DW 0000	DOS_ExeIP = 0
0000	DW 0000	DOS_ReloCS = 0
4000	DW 0040	DOS_TableOff = 40
0000	DW 0000	DOS_Overlay = 0
AA	DP AA	

Figure 2.0 Showing PE Dos Header comes here.

As shown in figure 2.0 the PE file format starts with a DOS stub with a header. The PE header is located at the offset 0x3C from the beginning of the file. The DOS header file is identified with the magic bytes "MZ" indicating that it is DOS header file. At the address 0x3C from the image base (or the address of the DOS header) is offset to the PE signature.

00	DB 00	
50 45 00 00	ASCII "PE"	PE signature (PE)
4C01	DW 014C	Machine = IMAGE_FILE_MACHINE_I386
0500	DW 0005	NumberOfSections = 5
D4C55548	DD 4855C5D4	TimeDateStamp = 4855C5D4
32100000	DD 00001032	PointerToSymbolTable = 1032
35010000	DD 00000135	NumberOfSymbols = 135 (309.)
E000	DW 00E0	SizeOfOptionalHeader = E0 (224.)
0201	DW 0102	Characteristics = EXECUTABLE_IMAGE 32B
0B01	DW 010B	MagicNumber = PE32
02	DB 02	MajorLinkerVersion = 2
37	DB 37	MinorLinkerVersion = 37 (55.)
00060000	DD 00000600	SizeOfCode = 600 (1536.)
00060000	DD 00000600	SizeOfInitializedData = 600 (1536.)
00020000	DD 00000200	SizeOfUninitializedData = 200 (512.)
25120000	DD 00001225	AddressOfEntryPoint = 1225
00100000	DD 00001000	BaseOfCode = 1000
00200000	DD 00002000	BaseOfData = 2000
00004000	DD 00400000	ImageBase = 400000
00100000	DD 00001000	SectionAlignment = 1000
00020000	DD 00000200	FileAlignment = 200
0100	DW 0001	MajorOSVersion = 1
0000	DW 0000	MinorOSVersion = 0
0000	DW 0000	MajorImageVersion = 0
0000	DW 0000	MinorImageVersion = 0
0400	DW 0004	MajorSubsystemVersion = 4
0000	DW 0000	MinorSubsystemVersion = 0
00000000	DD 00000000	Reserved

Figure 3.0 showing the PE header

As shown in figure 3.0, the field begins with the field PE\0\0 or | 50 45 00 00|. This header is followed by the COFF file header. The first field in the format is the machine type field indicated by the field *Machine* in figure 3.0. Its size is two bytes long. Value 0x8664 stands for AMD64, 0x14c stands for IA32 or less often 0x200 for IA64 Itanium processors. This field is followed by a two byte field which indicates the number of sections in the file. Its maximum value can be 96. Followed by this is the *TimeDateStamp* after which is the COFF symbol table. After this field, there is 4 byte field which indicates the number of symbols which are present. The last two fields stands for COFF fields of the size of two bytes, which indicate the size of the optional header and a field called the characteristics which defines the specific attributes to file. Its value determines if the file is a DLL, or a part of the system file or if it uses 32- bit words and so on. The optional header can be divided into three parts. The first eight fields are generic to the COFF. Following this is the 21 windows-specific fields, following which are data directories. The magic field represented as *MagicNumber* as shown in figure 3.0 can also be used to identify the PE version. The value of the field can be PE32 or PE32+. The size field represented by *SizeOfCode* as shown in the figure 3.0 displays the size of .text/code, .data and .bss section of the file. Following this is the field *AddressOfEntryPoint*, Address of the entry point or the address where the code will start executing. After this as shown in figure 3.0 is the base address of code and data. These addresses are specified by the field *BaseOfCode* and *BaseOfData*

00 00000000	SizeOfImage = 00000 (49152.)
00 00000400	SizeOfHeaders = 400 (1024.)
00 00000000	CheckSum = 0
00 0003	Subsystem = IMAGE_SUBSYSTEM_WINDOWS_CUI
00 0000	DLLCharacteristics = 0
00 00100000	SizeOfStackReserve = 100000 (1048576.)
00 00010000	SizeOfStackCommit = 1000 (4096.)
00 00100000	SizeOfHeapReserve = 100000 (1048576.)
00 00010000	SizeOfHeapCommit = 1000 (4096.)
00 00000000	LoaderFlags = 0
00 00000010	NumberOfRvaAndSizes = 10 (16.)
00 00000000	Export Table address = 0
00 00000000	Export Table size = 0
00 0000B000	Import Table address = B000
00 0000310	Import Table size = 310 (784.)
00 00000000	Resource Table address = 0
00 00000000	Resource Table size = 0
00 00000000	Exception Table address = 0
00 00000000	Exception Table size = 0
00 00000000	Certificate File pointer = 0
00 00000000	Certificate Table size = 0
00 00000000	Relocation Table address = 0
00 00000000	Relocation Table size = 0
00 0009000	Debug Data address = 9000
00 00000058	Debug Data size = 58 (88.)
00 00000000	Architecture Data address = 0
00 00000000	Architecture Data size = 0
00 00000000	Global Ptr address = 0
00 00000000	Must be 0
00 00000000	TLS Table address = 0
00 00000000	TLS Table size = 0
00 00000000	Load Config Table address = 0

Figure 4.0 Size of PE header continued.

The image base identified by the field *ImageBase* as shown in figure 3.0 specifies the address at which the image is loaded. The Microsoft document specifies this value to be a multiple of 64 K. The field “*Sizeof Image*” as shown in figure 4.0 specifies the total size of the file including size of file and all the headers. The field “*SizeofHeader*” specifies the size of the headers. The field “*DLLCharacteristics*” as shown in figure 4.0 is used for DLL and is used when the DLL is loaded. 0x0040 denotes the base address, 0x0080 indicates that the code integrity checks have been made, 0x0100 indicates that the image is no-execute compatible and 0x0400 denotes that the SHE (structured exception handling) is not used by the file.

Figure 5.0 shows the data directories. Other types of data are defined in the data directory.

```
NumberOfRvaAndSizes = 10 (16.)
Export Table address = 0
Export Table size = 0
Import Table address = 8000
Import Table size = 310 (784.)
Resource Table address = 0
Resource Table size = 0
Exception Table address = 0
Exception Table size = 0
Certificate File pointer = 0
Certificate Table size = 0
Relocation Table address = 0
Relocation Table size = 0
Debug Data address = 9000
Debug Data size = 58 (88.)
Architecture Data address = 0
Architecture Data size = 0
Global Ptr address = 0
Must be 0
TLS Table address = 0
```

Figure 5.0 Showing Export, Import and Resource Table

The export table as shown in figure 5.0 specifies the function exported by the file. Similarly the import table specifies the functions imported by the file. Resource table shown in figure 5.0 specifies the resources like icons used by the files whereas exception table indicates the registered exception used by the file. Base relocation in the file is specified in the base relocation table. Compiler generated debugging information is stored in the Debug data directory. Architecture is a reserved data directory and its value is set to zero. Global pointer data directory stores the RVA of the value to be stored in the global pointer register.

Name	Description	Associated Data Structure
Export Table	The table contains the name and RVAs of all the exported function in the current module	IMAGE_EXPORT_DIRECTORY
Import Table	It comprises of the list of module and function which is currently Imported	IMAGE_IMPORT_DESCRIPTOR
Resource Table	The table comprises of static definition or various user interface like strings, dialogue box and menus	IMAGE_RESOURCE_DIRECTORY
Base Relocation Table	It comprises of list of addresses which requires recalculation if the module gets relocated in any other address	IMAGE_BASE_RELOCATION
Debugging Information	Debugging Information for is contained in this table	IMAGE_DEBUG_DIRECTORY
Thread Storage Table	It contains the thread local variables. It is managed by the loader when the executable is loaded	IMAGE_TLS_DIRECTORY

Name	Description	Associated Data Structure
Load Configuration Table	This contains variety of the image configuration structure. They also contain information for special security feature which contains legitimate exception handler in the module	IMAGE_LOAD_CONFIG_DIRECTORY
Bound Import Table	They contain additional import related table which contains information on the bound import entries.	IMAGE_BOUND_IMPORT_DESCRIPTOR
Import Address Table	This comprises of list of functions imported from the current module. They are initialized in the load time to the actual address of the import functions.	This comprises of list of 32 bit pointers
Delay Import Descriptor	They contain information which can be used for implementing a delayed load importing mechanism. This is resolved when it is called. It is not supported by the operating system.	ImgDelayDesc

Figure showing Optional Directory in the Portable Executable File Format

```
.. TLS Table address = 0  
TLS Table size = 0  
Load Config Table address = 0  
Load Config Table size = 0  
Bound Import Table address = 0  
Bound Import Table size = 0  
Import Address Table address = 0  
Import Address Table size = 0  
Delay Import Descriptor address = 0  
Delay Import Descriptor size = 0  
COM+ Runtime Header address = 0  
Import Address Table size = 0  
Reserved  
Reserved  
.
```

Figure 6.0 showing TLS and and Import table

Thread local Storage (TLS) show in figure 6.0 data directory indicates the information used in the thread specific data storage. The local configuration table has different uses in different windows versions. XP is used to register the safeSEH function. The import address table (IAT) as shown in figure 6.0 is used to resolve the symbol address at run time. The reserved field must be set to 0.

The export table as mentioned above specifies the function exported by the file. The export table is specified in the .edata section. The export directory table describes the entirety of the export information. It comprises information which can be used to resolve imports to the exported functions within the image. The export address table comprises the address of the exported entry points data and absolutes. The name pointer table consists of the arrays of RVAs into the export name table. An ordinal number is used to index the export address table. The Ordinal Base must be subtracted from the ordinal number to index into the table. The ordinal table as shown in the figure comprises an array of 16-bit ordinals into the export address table. The Export Name Table Pointers and the Export Ordinal Table form two parallel arrays. The exported address table ordinal numbers corresponding to the named export referenced by corresponding export name table pointer is in the export ordinal table array. The export name table comprises the null terminated variable length string names of exported functions/data/etc. It comprises the ASCII names for exported entries in the image. The Export Name Table Pointers and the array of Export Ordinals along with the Export Name table are used to translate a procedure name string into ordinal number. This is performed by searching for a matching name string. Entry point information in the export address table is located by the ordinal number.

Offset	Size	Field Name
0	4	Export Flags
4	4	Time/ Date Stamp
8	2	Major Version
10	2	Minor Version
12	4	Name RVA
16	4	Ordinal Base
20	4	Address Table Entries
24	4	Number of Name Pointers
28	4	Export Address Table RVA
32	4	Name Pointer RVA
36	4	Ordinal Table RVA

Figure 7.0 Showing Export Directory Table

As shown in the figure 7.0, the export directory table contains the Export Flags, Time/Data Stamp, Major version/ Minor Version, Name RVA, Ordinal base, Address Table RVA, Name PTR table RVA, Ordinal table RVA. Name RVA is the relative virtual address of the DLL ASCII Name. This is the address relative to the Image Base. Ordinal base is typically set to 1. This field specifies the starting ordinal number for the export address table for the image. The field addresses table entries and the number of name pointers denotes the number of entries in the address table and name table. Address Table RVA denotes the relative virtual address of the export address table. This is relative to the Image base. Name Table RVA contains the virtual address of the export name table pointers relative to the beginning of the Image base. Ordinal table RVA contains the relative virtual address of export ordinal table entry relative to the beginning of the Image base. It is an array of 16 bit indexes biased by the ordinal base into EAT. A symbol can be resolved by using the following steps.

1. VA or the export directory table in the optional header has to be obtained.
2. This VA is then used to locate the ordinal base, export directory table and the ordinal RVAs.
3. Obtain the RVA of the name pointer RVA.
4. It has to be determined if the function is exported by name. This is done by searching the export name table pointer.
5. Ordinal is then retrieved. This is done by using the index into the name pointer table as an index into the ordinal table to retrieve the ordinal.

6. Ordinal is taken and subtracted from the ordinal base and the result is used as an index into the EAT. Data at this index is the RVA for the exported function.

The import data table is the same as the export data table. It uses import table or .idata section similar to the export table. “Import Directory Table”, “Import Lookup Table” and the “hint/name table” are the three structures used for importing symbols

### 3.2. Import Address Table

Windows loader is responsible for reading in PE file structure and loading the executable image in memory. Windows loader also loads all the .dll files that an application uses and is responsible for mapping them into the address space. The executable will require functions whose addresses are not static. Import table, comprising function pointer, is used to get the addresses of the functions when the dlls are loaded. It can be accessed either by the call [pointer address] or by “Import Lookup Table” and the “hint/name table,” which are the three structures used for importing symbol. Import directory table uses import table or .idata section.

Offset	Size	Field Name
0	4	Import Lookup Table RVA
4	4	Time/Date Stamp
8	4	Forward Chain
12	4	Name RVA
16	4	Import Address Table RVA

Figure 8.0 Showing Import Table

The Import Directory Table (shown in figure 8) comprises array of Import Directory Entries, one entry of each DLL. The last directory entry is identified by the NULL specifies the date and time when the import data was presnapped or zero if not pre snapped, which denotes the end of the directory table. Import flags are set to 0. Major and minor version field represents the major and minor version of the dll being referenced. Name RVA field specifies the relative virtual address relative to the Image base. Import Lookup Table RVA contains the address relative to the beginning of the image base, of the start of the import lookup for the image. Import Lookup Table is an array of 32-bit integer, comprising bit field entry of ordinal or hint/name RVA’s for each DLL. Last entry is indicated by the NULL.

Whether the import is done by name or by ordinal is indicated by the high order bit. The set value of the bit indicates that it is imported by ordinal and the bits 0 to 15 indicate the ordinals to import. When imported by name, the bits 0 to 30 represent a 31-bit RVA into the hints/ name table for the name of the imports.

Bits	Size	Field Name
1	1	Ordinal/Name
15 – 0	16	Ordinal Number
30 - 0	31	Hint Name Table RVA

Figure 9.0 Showing the Import Look up table

As shown in figure 9.0 in the Hint-Name Table, the PAD field is optional. HINT is the DW hint into the export name table pointer. The value is used to index the export name table pointer array, allowing faster ‘by name’ imports. The field ASCII string as shown in the figure is terminated by NULL bytes.

Offset	Size	Field Name
0	4	Characteristics
4	4	TimeDateStamp
8	2	MajorVersion
10	2	MinorVersion
12	4	GlobalFlagsClear
16	4	GlobalFlagsSet
20	4	CriticalSectionDefaultTimeout
24	8	DeCommitFreeBlockThreshold
32	8	DeCommitTotalFreeThreshold
40	8	LockPrexitTable
48	8	MaximumAllocateSize
56	8	VirtualMemoryThreshold
64	8	Process Affinity Mask
72	4	Process Heap Flag
76	2	CSD Version
78	2	Reserved
80	8	Edit list
88	4	Security Cookie
96	4	SEHandlerTable
104	4	SEHandlerCount

Figure 10.0 showing the load configuration structure

Figure 10.0 shows the load configuration structure. Valid exception handler is registered by the system. This prevents attackers from overwriting an SHE entry and causing an exception to be raised and their code executed.

The “SE handler count” field in the “load configuration structure” denotes the count of total number of handlers. The field “SE handler table” is a sorted table of RVA, which corresponds to valid SHE handler for that image. The security cookie is a pointer to a cookie. In Microsoft compiler, when the GS flags are set the cookie, the stack bases cookie is used to prevent the stackbased overflow. The value of the field “IMAGE\_DLLCHARACTERISTICS\_NO\_SEH” in the DLL characteristics field of optional header specifies if the exception handler is in the list. If the value is set then it denotes that the exception handler is in this list.

### **3.3 Executable and Linking Format**

ELF (executable and linking format) is the default binary format on operating systems such as Linux, Solaris and SVR4. It provides the capability of dynamic linking, dynamic loading, imposing run time control on the program and improved method of creating shared libraries. ELF file format enables identification and parsing of object files on different platforms. Executable, relocatable and shared objects are the different type of ELF files. They store code, data and information about the program, which aids operating system in performing actions on these files. An executable file contains the information which is required for the operating system to create a process image. The process image is required for accessing the data and executing the code. Linking with other object files to create and executable file is done by the relocatable file. Information required for static and dynamic linking is stored in shared object file. ELF file format includes five sections. (1) ELF header (2) The program header (3) The section header table (4) The ELF sections (5) The ELF segments.

#### **3.3.1 ELF Header**

It is first section and at fixed position in the object file. The other headers may or may not be present in the file. The header aids in identifying if the object file is relocatable, executable, shared or core file. The header also provides information about the program header table, Section header table and String table. It also provides the associated numbers and the size entries for each table. Location of first executable instruction is also located in the ELF header. Figure 11.0 shows the ELF header

```
typedef struct
    unsigned char e`ident[EI_NIDENT];
    Elf32` Half e`type;
    Elf32` Half e`machine;
    Elf32` Word e`version;
    Elf32` Addr e`entry;
    Elf32` Off e`phoff;
    Elf32` Off e`shoff;
    Elf32` Word e`flags;
    Elf32` Half e`ehsize;
    Elf32` Half e`phentsize;
    Elf32` Half e`phnum;
    Elf32` Half e`shentsize;
    Elf32` Half e`shnum;
    Elf32` Half e`shstrndx;
    Elf32` Ehdr;
```

Figure 11.0 Showing the ELF Header.

```
typedef struct
    Elf32` Word P type;           // type of Segment
    Elf32` Off p` Offset;        // File offset to Segment
    Elf32` Addr p` vaddr ; // Virtual Address of First Byte
    Elf32` Addr P` paddr; // Segment physical Address
    Elf32` Word p` filesz; // Size of File Image Segment
    Elf32` Word p` memsz; //Size of Memory Segment
    Elf32` Word p` flags; // Segment- Specific Flag
    Elf32` Word p` align; // Alignment requirement
    Elf32` Phdr ;
```

Program Header

Figure 12.0 Showing Program Header

### 3.3.2 The Program Header Table

Program headers comprise a series of array where each entry is a structure. The structure describes the segment in the object file or other information required to create an executable process image. ELF header consists of the size and number of entries in the table. Type, file offset, physical address, virtual address, file size, memory image size, and alignment are contained in each entry in the program header table. Process image for the object file is created by the program header. The *p\_type* field is shown in 12.0. If the value of *p\_type* is PT\_LOAD, the operating system copies the segment into memory according to the location and size information. If *p\_memsz* is greater than

these *p\_filesz* then these bytes are mapped into the segment. PT\_LOAD segment is succeeded by the PT\_INTERP segment. PT\_DYNAMIC segment is related with the dynamic linking. PT\_INTERP segment denotes the path name of the program interpreter. PT\_DYNAMIC segment is related to dynamic linking. If a file contains segment PT\_SHLIB, then it does not conform to ABI. It is defined but reserved. Segment of type PT\_PHDR indicates the size and location of the program header table. This is applicable both in physical and in the memory image. It can appear only once and it should occur before PT\_LOAD segment. PT\_LPROC and PT\_HIPROC are reserved for processor-specific functionality.

The member *p\_offset* specifies the offset of the segment from the beginning of the file and the member *p\_vaddr* denotes the preferred virtual address of the segment. The member *p\_filesz* indicates the size of segment in the physical file and the member *p\_memsz* denotes the size of segment in memory. *P\_flags* denote the attributes of the segment. To load an executable, address for each segment specified in the *p\_vaddr* is used. Images have absolute references. If the address is changed it will break. In the case of ASLR images and shared library, make use of position-independent code. In case of PIC, instead of using absolute references, relative references are used. When shared library is used to access commonly used functions, series of intermediaries are used in the case of ELF. They are global offset table (GOT or .got), the dynamic segment/section (\_DYNAMIC or .dynamic) and the procedure linkage table (PLT or .plt). Every executable image which performs dynamic linking contains segment .dynamic.

Offset	Size	Field Name
0	4	d_tag
4	4	d_tag
4	4	D_ptr

Figure 13.0 Showing Dynamic Structures

It contains two values, a tag followed by a union. The tag determines the interpretation of union. The member *d\_val* comprises integer with various interpretations. The member *d\_ptr* contains a VA. The figure 14.0 shows different types of *d\_tag* types, and whether they are optional or mandatory.

\_DNAMIC array is ended by DT\_NULL. DT\_PLTRELSZ element holds the total size of the relocation entries associated with the PLT. DT\_PLTESZ entry

Name	Value	d_val or d_ptr?	executable	Shared Object
DT_NULL	0	ignored	mandatory	mandatory
DT_PLTRELSZ	2	d_val	optional	optional
DT_PLTGOT	3	d_ptr	optional	optional
DT_FINI	13	d_ptr	optional	optional
DT_PLTREL	20	d_val	optional	optional
DT_JMPREL	23	d_ptr	optional	optional

Figure 14.0 showing defined d\_tag Types

is present when DT\_JMPREL is present. A DT\_PLTGOT entry contains the address corresponding with either the GOT or PLT. The DT\_JMPREL entry comprises a pointer to relocation entries associated only with the PLT. During image initialization, these relocations are ignored and it uses a form of linking known as lazy binding. As per the lazy binding, the relocation is ignored until the actual use of symbols. There is increase in speed and efficiency and dynamic module can be easily loaded.

\_GLOBAL\_OFFSET\_TABLE comprises array of addresses, which are absolute references and allow the position-independent code to have relative address. Negative and positive indexes are both valid since, the symbol \_GLOBAL\_OFFSET\_TABLE does not need to refer to the beginning of the .got segment. Dynamic linker processes the GOT by using the first entry which is first element which contains the address of the \_DYNAMIC structure. Position-independent addresses are redirected to absolute locations by the GOT, PLT performs for the same for functions. It determines the absolute address of the function and updates GOT as necessary. The second and the third entries in the GOT, upon the creation of image are defined as follows.

The address of the GOT must reside in the ebx, if the PLT is PIC. The calling function places the address into this register. The application is trying to call *fun* which is located in the label PLT1. The first instruction under that label is a jump into the GOT, which contains the address of the *push* and *jmp* instructions. Variable name *offset* will specify the GOT entry used in the prior jump along with a symbol table index *fun* in this instance. The application then jumps to PLT0 and pushes the address of the second element of the GT onto the stack. This provides dynamic linker a word to reference for identification purposes. It then transfers control to the third GOT entry which transfers control to the dynamic linker. The stack is then un-winded and it then retrieves the identifying information and finds the absolute address for the symbol. The absolute address is stored in the related GOT entry and the

control is handed over to the requested function. Dynamic linking is accomplished by indirection and abstraction because, any further calls to this function will jump directly to PLT0 since the GOT entry is modified. Application makes a call to PLT when it does not know before hand what address it is calling. If the address has not been resolved then the it calls PLT, then jumps to the GOT, if the address has not already been resolved, control is handed back into the PLT if the address has not been resolved. The relocation entry is then pushed followed by jump to the first entry in the PLT, which then hands control to the dynamic linker.

### 3.4 Conclusion

This chapter presents the details of PE file format. The file format comprises file headers, data directory, section table and various other sections. It provides the details of import and export tables, Export directory, import directory, relocation table and debug directory and the thread local storage. Symbols imported in the file are loaded in the import table while the export table lists all the symbols imported by the PE file. Further details of PE specification can be obtained from Microsoft's website at [www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx](http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx).

Though PE file format is one of the important file formats, Appendix lists some of the other file formats along with their hex signature which can be used by IDS/IPS rules to identify the file. The appendix provides the extension of the other file formats.

# Reversing Binaries for Identifying Vulnerabilities

## 4.0 Introduction

Vulnerability is defined as a bug or flaw in an application which allows malicious intruders to compromise the system. In the case of remote code execution vulnerability, the application takes the user input which can be a command line parameter that a program receives, a file loaded into the program, or a packet sent over the network. The input is then used to make a program stray from its normal execution path. One of the simplest methods to exploit vulnerability is to make the program crash. However shell codes injected by the user input can be used to take control of a program.

This chapter discusses various vulnerabilities in an application. By using case studies of real exploits it discusses the exploitation of vulnerabilities. The chapter also discusses the identification of these vulnerabilities by analyzing assembly code.

## 4.1 Stack Overflow

Buffer can be defined as a contiguous chunk of memory, which consists of an array or pointer in C. In C code if no bound checking takes place, then a user can write past the buffer.

```
#include<stdio.h>

void main(){
    int buffer[5];

    buffer[10] = 5;
}
```

Figure 1.0 showing the C code attempting to write past the buffer

Even though the code shown in figure 1.0 is valid code, it can be seen that the program attempts to write past the allocated buffer, which will trigger unexpected behavior. A stack is generally used whenever a function call is made. A stack is a contiguous block of memory containing data. Stack pointer

point to the top of the data. As shown in figure 3.0, function parameters are pushed from right to left and are followed by pushing the return address (return address is the address which needs to be executed when a function returns.), and frame pointer (FP) on to the stack. A frame pointer is used as a reference to the local variables and to the function parameters. Local automatic variables are pushed after the FP

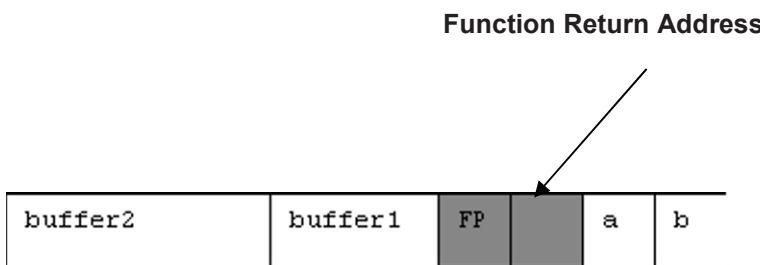
---

```
#include<stdio.h>

void foo(int a, int b) {
    char buffer1[10];
    char buffer2[10];
    strcpy(buffer1, "I am overflowing Buffer\n");
}

void main(){
    foo(10,20)
}
```

**Figure 2.0 Showing the C code using Buffers**



**Bottom of Memory**

**Top of Memory**

**Figure 3.0 showing the stack function for C code in 11.0**

The program shown in figure 2.0 is guaranteed to cause unexpected behavior as a string of length greater than 10 has been copied to a buffer that has been allocated 10 bytes.

<pre> . 55      PUSH EBP . 89E5    MOV EBP,ESP . 83EC 0C  SUB ESP,0C . B9 03000000 MOV ECX,3 &gt; 49      DEC ECX . C7048C 5A5AF9 MOV DWORD PTR SS:[ESP+ECX*4],FFFFA5A5 .^75 F6    JNC SHORT bo.004012DF . 57      PUSH EDI . 68 A3004000 PUSH bo.0040A0A3 . 8D70 F6    LEA EDI,DWORD PTR SS:[EBP-A] . 57      PUSH EDI . E8 152E0000 CALL bo.00404100 . 83C4 08    ADD ESP,8 . 8D70 F6    LEA EDI,DWORD PTR SS:[EBP-A] . 57      PUSH EDI . 68 A0004000 PUSH bo.0040A0A0 . E8 D85E0000 CALL bo.004071E1 . 83C4 08    ADD ESP,8 . 5F      POP EDI . C9      LEAVE . C3      RETN \$ 55      PUSH EBP . 89E5    MOV EBP,ESP . C9      LEAVE . C3      RETN \$ 53      PUSH EBX </pre>	<pre> ASCII "I am going to Overflow Buffer" [Arg2 Arg1 = 0040A0A0 ASCII "%s" bo.004071E1 </pre>
--	---

Figure 4.0 showing the assembly for the C code shown in figure 2.0

These extra bytes will run past the buffer, and will overwrite the space, which has been allocated for FP, return address and so on. The extra bytes corrupt the process stack and overwrite the functions' return address. The code, which must be executed, should be placed in the buffer's overflowing area, and hence by overwriting the function's return address; the intended code can be executed. Figure 4.0 shows the assembly instruction for the function *foo* of the C code shown in figure 2.0. While analyzing the assembly for stack overflow vulnerability several patterns need to be searched. Stack size has to be monitored first. This can be identified by instruction SUB ESP at the beginning of program. As shown in figure 4.0 a SUB ESP, 0C exists. Functions will have a large local variable space in stack, as they take in a large buffer and put it on the stack. Once the size of buffer is determined, the next step will be to identify the pointer to the beginning of that space. This generally can be identified by searching for a LEA instruction. For example for the instruction LEA, if the operand is [ESP - 0x15] or [EBP - 0x20], then the constant should be equivalent to the space being allocated. As shown in figure 4.0, the string "I am overflowing buffer" gets which is more than 10 bytes gets copied in the allocated space. (as shown in figure 2.0) Mostly overflow attacks make use of string manipulation routines. For example, strcpy stops copying when the NULL terminator character is encountered. It might happen that the caller will supply a string which is long resulting in an overflow.

#### **4.1.1 CAN-2002-1123 Microsoft SQL Server 'Hello' Authentication Buffer Overflow".**

For this vulnerability, the vulnerability exists in the SSnetlib.dll library. The assembly code is from memory on a Windows 2000 machine with Microsoft SQL server 2000 SP2. Figure 5.0 shows the malicious assembly code.

```

JE SHORT SSNETLIB.xxxxxxxx
JMP SSNETLIB.yyyyyyyy
MOV EAX,DWORD PTR SS:[EBP+C]
ADD EAX,DWORD PTR SS:[EBP-218]
PUSH EAX
LEA ECX,DWORD PTR SS:[EBP-214]
PUSH ECX
CALL <JMP.&MSVCRT.strcpy>
ADD ESP,8
PUSH SSNETLIB.zzzzzzzz
LEA EDX,DWORD PTR SS:[EBP-214]
PUSH EDX
CALL <JMP.&MSVCRT.strcmp>
```

Figure 5.0 showing the assembly instruction for the Ssnetlib.dll.

The assembly instruction CALL <JMP.&MSVCRT.strcpy> has two parameters. The first parameter contains the destination address to which the data at the address has to be copied; the second parameter refers to the memory location which contains the TDS payload. The payload is copied into the memory on stack which is referenced by the first parameter. As seen from the assembly instruction, the size of the destination buffer is 512 bytes. If the size of buffer is greater than 512 bytes, an access violation interrupt will be raised. Also, if the size of the buffer is specified to be small, 0 or NULL bytes, a large payload will overwrite the return address.

#### **4.1.2 CAN -2004-0399 Exim Buffer Overflow**

For CAN-2004-0399, vulnerability existed in processing the sender's address when it is supplied to exit by an SMTP command "MAIL FROM: sender\_address". From the source code analysis shown in 6.0, if the sender's email address exceeds 256 bytes, it will result in overflow of fixed size stack buffer "buffer". The function sprintf() will result in buffer overflow.

Exim version: 3.36, file: *verify.c*, function: *BOOL verify\_sender(int \*errcode, char \*\*errmess)*:

```
BOOL verify_sender(int *errcode, char **errmess) {
...
char buffer[256];
...
sprintf(buffer, "%s:%.200s", sender_address,
(sender_host_name != NULL) ? sender_host_name :
7
(sender_host_address != NULL) ? sender_host_address : "");
```

Figure 6.0 showing the vulnerable code

“Exim Header Syntax Checking Remote Stack Buffer Overrun Vulnerability”, happens when the vulnerable process copies a malicious string to a fixed size buffer of 64. The code for the function which can overflow is shown in figure 6.0. The length of the string is not checked. Headers which can trigger the vulnerability are “from”, “sender”, “reply-to”, “to”, “cc” and “bcc” followed by enough spaces to overflow the 64 byte buffer.

**Exim version: 3.36, file accept.c, function: *BOOL accept\_msg(FILE \*fin, BOOL extract\_recip)***

```
BOOL accept_msg(FILE *fin, BOOL extract_recip)
...
if (headers_check_syntax)
{
...
if (recipient == NULL && strcmp(errmess, "empty address") != 0)
{ char hname[64];
char *t = h->text;
char *tt = hname;
char *verb = "is";
int len;
while (*t != ':') *tt++ = *t++;
*t = 0;
...

```

**Exim version: 4.32, file verify.c, function: *BOOL verify\_check\_headers()***

```
int
verify_check_headers(uschar **msgptr)
{
header_line *h;
uschar *colon, *s;
for (h = header_list; h != NULL; h = h->next)
{
if (h->type != htype_from &&
h->type != htype_reply_to &&
h->type != htype_sender &&
h->type != htype_to &&
h->type != htype_cc &&
h->type != htype_bcc)
continue;
...
if (recipient == NULL && Ustrcmp(errmess, "empty address") != 0)
{ uschar hname[64];
uschar *t = h->text;
uschar *tt = hname;
uschar *verb = US"is";
int len;
while (*t != ':') *tt++ = *t++;
*t = 0;
```

Figure 7.0 showing the vulnerable code for Exim Header Syntax Checking Buffer Overflow.

### 4.1.3 Stack Checking

The latest version of the MS compiler provides protection against stack overflow attacks. They achieve stack overflow protection by placing a cookie between the last local variable and the function's return value. This cookie should be validated before the function's return address. In case of buffer overflow the value of the cookie will be modified and the execution of the program will stop. The cookie for the Windows operating system is placed in a protected module (`_security_cookie`). The module is initialized by the `_security_init_cookie` when the module is loaded and is randomized. The randomization is based on the current process and the thread ID's along with the current time.

## 4.2 Off-by-One Overflow

In Off-by-One overflows, the operation is performed on a buffer which is allocated by that size. The string in order to terminate must include a null byte terminator. It might happen that the string is not terminated by a null terminator. This will result in a string to edge with another buffer on the stack. In such a scenario, there will be no separation, between the strings hence causing an overflow. For example, as shown in figure 8.0, the size of the buffer is 300. In the function `foo` three hundred and one 'A' are copied in the buffer. This results in overwriting the null character and hence results in an Off-by-One Overflow.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void foo(char *name){

    char buffer[300];
    int a;
    for(a=0;a<=300;a++)
        buffer[a] = name[a];
}

void main() {
    char name[400];
    int i=0;
    for(i=0;i<400;i++)
        name[i]='A';
    foo(name);
}
```

Figure 8.0 C code showing Off-by-one error

The assembly instruction for the function *foo* in the code in figure 8.0 is shown in figure 9.0. In the assembly code it can be seen that memory is first allocated then it is filled with FFFA5A5A. Figure 10.0 shows the memory map. The address 0012FE2C holds the value of the counter. As it can be seen from the assembly instructions in figure 10.0, the instructions

```
INC DWORD PTR SS:[EBP+EDI-130],BL
CMP DWORD PTR SS:[EBP-4],12C
```

increments the value at the counter at address 0012FE2C , and compares it with hexa 12C equivalent to decimal 300. If the value of the counter is less than 300, the A is copied by using the instruction

```
MOV BYTE PTR SS:[EBP+EDI-130], BL
```

```
PUSH EBP
MOV EBP,ESP
SUB ESP,130
MOV ECX,4C
DEC ECX
MOU DWORD PTR SS:[ESP+ECX*4],FFFFA5A5A
JNZ SHORT testvr.004012E2
PUSH EBX
PUSH ESI
PUSH EDI
MOV DWORD PTR SS:[EBP-4],0
MOV EDI,DWORD PTR SS:[EBP-4]
MOV ESI,DWORD PTR SS:[EBP+8]
MOV BL,BYTE PTR DS:[ESI+EDI]
MOV BYTE PTR SS:[EBP+EDI-130],BL
INC DWORD PTR SS:[EBP-4]
CMP DWORD PTR SS:[EBP-4],12C
JLE SHORT testvr.004012F6
POP EDI
POP ESI
POP EBX
LEAVE
RETN
DISM CDD
```

Figure 9.0 Showing the assembly language for the C code shown in the figure 8.0

0012FE10	FFFFA5A5A
0012FE1C	FFFFA5A5A
0012FE20	FFFFA5A5A
0012FE24	FFFFA5A5A
0012FE28	FFFFA5A5A
0012FE2C	000000C5
0012FE30	0012FF70
0012FE34	00401361 RETURN to testvr._main+4A from testvr._foo
0012FE38	0012FE40 ASCII "AAAAAAAAAAAAAAAAAAAAAAFAF
0012FE3C	7C910798 ntdll.7C910798
0012FE40	41414141
0012FE44	41414141

Figure 10.0 Memory map showing the FFFA5A5A in the allocated space and counter at address 0012FE2C.

The figure 11.0 shows the memory map, for the function *foo* shown in figure 8.0, when the value of counter *a* reaches 300. As per the memory map, the allocated buffer of size 300 is filled with A.

0012FE24	41414141
0012FE28	41414141
0012FE2C	0000012C
0012FE30	0012FF70
0012FE34	00401361 RETURN to testvr._main+4A from testvr._foo
0012FE38	0012FE40 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAF
0012FE3C	7C910738 ntdll.7C910738
0012FE40	41414141

Figure 11.0 Memory map showing the FFFA5A5A in the allocated space and counter holding value of 12C at address 0012FE2C.

However, as in the function *foo* shown in figure 12.0 the loop gets executed 301 times. The last execution of the loop results in rewriting at the address space 0012FDC8. As shown in the figure, the value 12C is overwritten with 141. This is due to the Off-by-One error. The value at the address is overwritten, and as the value 141 is less than 12C, the loop terminates. However, by carefully crafting exploits, it might be possible to rewrite other sections of memory.

0012FDC0	41414141
0012FDC4	41414141
0012FDC8	00000141
0012FDCC	0012FF70
0012FDD0	00401361 RETURN to testvr._main+4A from testvr._foo
0012FDD4	0012FDDC ASCII "AAAAAAAAAAAAAAAAAAAAAAAF
0012FDD8	7C910738 ntdll.7C910738
0012FDDC	41414141

In figure 12.0 ,Memory map showing the 41 in the allocated space and counter at address 0012FE2C is overwritten due to the off by one error.

Sometimes it might happen that the buffers are allocated on the stack, however, it might not copy enough data to cause an overflow. It might not be possible to rewrite via the *ret*. In such a scenario, it is possible to rewrite the other important data like EBP which might be used later. In Off by One overflow, overflows comprise overflowing the buffers adjacent to stored EBP on the stack in a called function. This creates a fake frame for the caller function. Hence, when the caller function exits, it is forced to use the return address supplied by an attacker in an overflowed buffer or somewhere else in memory.

### 4.2.1 OpenBSD 2.7 FTP Daemon Off-by-One

A buffer overflow was discovered in OpenBSD distribution in a piece of code handling directory names. The vulnerable piece of code is shown in figure 13.0

*MAXPATHLEN* is defined in *<sys/param.h>*, and is of size 1024 bytes. The *for()* loop correctly bounds variable *i* to < 1023, such that when the loop has ended, no byte past *npath[1023]* should be written with \0. However, since *i* is also incremented in the nested statements as *++i* it becomes equal to 1024, and *npath[1024]* is past the end of the allocated buffer space and a null byte is written into *npath[1024]*, overwriting the least significant byte of EBP.

```
replydirname(name, message)
{ const char *name, *message;

    char npath[MAXPATHLEN];
    int i;
    for (i = 0; *name != '\0' && i < sizeof(npath) - 1; i++, name++) {
        npath[i] = *name;
        if (*name == '\\')
            npath[++i] = '\\';
    }
    npath[i] = '\0';
    reply(257, "%"s"\n%s", npath, message);
}
```

Figure 13.0 Vulnerable code in OpenBSD FTP Daemon

This can be exploited as an off-by-one overflow. The vulnerability was fixed using the code shown in figure 14.0 .

---

```
replydirname(name, message) {
    const char *name, *message;
    char *p, *ep;
    char npath[MAXPATHLEN];
    p = npath;
    ep = &npath[sizeof(npath) - 1];
    while (*name) {
        if (*name == '\\' && ep - p >= 2) {
            *p++ = *name++;
            *p++ = '\\';
        } else if (ep - p >= 1)
            *p++ = *name++;
        else
            break;
    }
    *p = '\0';
    reply(257, "%"s"\n%s", npath, message);
}
```

Figure 14.0 Modified Code with no Off-by-One Vulnerability

The pointers  $p$  and  $ep$  guarantee that the closing quotation mark is inserted only if the end of the buffer  $npPath[1023]$  has not been achieved yet. Pointer  $p$  is also always less than  $ep$ , and is not greater than  $\&npPath[sizeof(npPath)] - 1$ . Hence when the code  $*p = '\0';$  is executed, the null byte is never written past the allocated space.

### 4.2.3 Non-Executable Memory

Non-executable memory can also be used to prevent buffer overflow. There are certain memory pages which are defined to be non-executable by the processor. Non-executable memory pages can be used to store data. The processor will not execute code stored in these memory pages. The operating system can label stack and data pages as non-executable which will prevent an attacker from running code on them using buffer overflow. Recent versions of Intel, AMD processors, IA-64 processors provide support for non-executable memory. Windows XP service pack 2 and above, Solaris 2,6 and above and Linux provide support for non-executable memory.

It might be possible to exploit the system having non-executable memory. One of technique used is return-to-libc. By using this technique, the function's return address is pointed to a well known, function, (runtime library function or a system API) which will enable access to the process. This technique defeats non-executable stacks. However it requires complicated exploits.

## 4.3 Heap Overflows

Heap is used from dynamic allocation of data. Address space is allocated in the same segment as stack; however, it grows towards a stack from a higher address to a lower address.

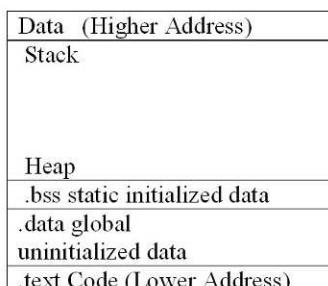


Figure 15.0 Showing the memory arrangement.

Memory is allocated by the malloc functions like *malloc()*, *HeapAlloc()*, *new()*. As shown in figure 16.0, the buffers *buffer\_overflow*, *buffer\_normal* are allocated to size 15. The function *strcpy* is used to copy the string “normal buffer” to buffer *buffer\_normal*. In the next instruction, an overlong string is copied in the buffer *buffer\_overflow*. As shown in figure 17.0, copying overly long strings in the buffer *buffer\_overflow* results in them overflowing the allocated heap *buffer\_normal*, and they are copied with the *aaaaaa*. The heap overflow does not necessarily result in crashing of an executable. Similar kinds of overflow can be executed on the stack variables, which are located in the BSS segment.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void main(){

    char *buffer_overflow = malloc(15);
    char *buffer_normal = malloc(15);

    strcpy(buffer_normal,"normal buffer");
    strcpy(buffer_overflow,"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");

    printf("The normal buffer at %p is %s\n", buffer_normal,buffer_normal);
    printf("The overflow buffer at %p is %s", buffer_overflow,buffer_overflow);

}
```

Figure 16.0 showing C code of heap Overflow

Heaps are arranged as a linked list. In heaps, the pointers to the next and previous heap are placed before and after the actual block of data. Heap Overflow or the malloc exploit belongs to the other class of exploits which can be used to take control of the program. In heap-based overflow, the program receives data of unexpected lengths which are copied into a small size buffer.

```
The normal buffer at 0x00146120 is aaaaaaaaaaaaaaaa♦
The overflow buffer at 0x00146108 is aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa♦
"C:\lcc\lcc8\testheap.exe"
Return code 78
Execution time 0.032 seconds
Press any key to continue...
```

Figure 17.0 Output of the C code shown in figure

This will result in rewriting the address of the block following the heap block in memory. When the heap manager traverses the modified linked list it will result in the program crashing. It has to be noted in heap-based overflow that the Heap grows upwards or the new variable which is created in the heap is located at the higher memory address. Hence the buffer which overflowed is located lower on the heap. As shown in figure18.0, heap allocation consists of a minimum of eight block size. Besides this there is an additional overhead of eight block. This is known as a heap control block.

<b>Size of this Block /8</b>	<b>2 Bytes</b>
<b>Size of previous Block / 8</b>	<b>2 Bytes</b>
<b>Flags (8 Bit)</b>	<b>4 Bytes</b>
<b>DATA if in use else previous free block pointer</b>	<b>4 Bytes (EAX)</b>
<b>DATA if in use, else next block pointer</b>	<b>4 Bytes (ECX)</b>

Figure 18.0 showing the layout of Heap.

Besides crashing a program as each block has pointer to “next” and “prev” it might be possible to overwrite memory in address space. To prevent heap based overflow the canary value has to be placed between all the variables, on the heap space. This canary value must not be altered through out the execution of a program.

### 4.3.1 Heap Based Overflows

Linux OS makes use of *dlmalloc*. The heap memory is organized into chunks. The chunks contain the information which is used by *dlmalloc* to allocate free memory efficiently.



Figure 32.0 Heap Memory from *dlmalloc*'s view.

The value of the *prev\_size* element contains the size of the chunk of the previous to the current one. The value exists only when the chunk before it is unallocated. Under this scenario, when the chunk of the previous to current one is allocated, the *prev\_size* is not used and is used by the data element to save four bytes. The size of the currently allocated chunk is stored in the *size* element. It has to be noted that four is added to the length argument and it is then rounded to the next double word boundary when the function call *malloc* is called. It might happen that some of the bits are set to zero due to rounding, in such a case, *dmalloc* uses them as flags for attributes on the current chunk. For exploitation of a heap, the lowest most bit is important. This bit holds the value of *PREV\_INUSE* flag, which indicates whether the previous chunk is allocated or not. The element *data* denotes the space allocated by the *malloc()* returned as a pointer. In this space, data used by the application is copied. When the *free* function call is made, the *dmalloc* first checks if the neighboring blocks are free. If the neighboring blocks are free, then the neighboring chunk is merged with the current chunk providing a large block of free memory.

Prev size
Size
fd
bk
Unused Memory

Figure 33.0 showing freed *dmalloc* block.

The fd and bk pointers which stand for forward and backward replace the first eight bytes of the previously used memory. Whenever *free* is called, a check is made to ensure if there are any unallocated chunks. The unused memory is the old memory which was in the chunk. It has to be noted that in *dmalloc*, management information for the memory chunks is stored in-band with the data. When a chunk of memory is unallocated using *free()*, then the chunk is checked to ensure that it borders the top-most chunk. It might happen that the chunk which is being freed is set to “not-in-use”. In such a scenario, the precious chunk is taken from the linked list. It is then merged with the chunk which is currently free.

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void main(int argc, char**argv){

    char *buffer_overflow = malloc(200);
    char *buffer_normal = malloc(150);

    strcpy(buffer_overflow, argv[1]);

    free(buffer_overflow);
    free(buffer_normal);

}

```

Figure 33.0 showing a vulnerable program using *malloc*.

In figure 33.0, strcpy is performed without bounds checking into the *buffer\_overflow* buffer. The pointer *\*buffer\_overflow* points to 200 bytes of memory. If the input is more than 200 bytes, then it will overflow *\*buffer\_normal* as the two chunks are adjacent in memory. The *prev\_size*, *size* and *data* of the *buffer\_normal* will be overwritten. This vulnerability can be exploited by crafting a malicious chunk of *fd* and *bk* pointers which control the order of the linked list. The *fd* and *bk* pointers will have to be changed for the address to be overwritten with the address of the exploit code. It might happen that the overflowed chunk can be at the border of the top-most chunk. In such a scenario, a macro unlink can be used.

---

```

#define P (next)
#define unlink(P)|
{
*(next->bk + 8) = P->bk; \
*(next->fd + 12) = P->fd; \
*(next->fd + 12)->bk = *(next->bk + 8); \
*(next->bk + 8)->fd = *(next->fd + 12); \
}

```

Figure 34.0 showing psuedo code for unlink macro.

As the value of the *bk* and the *fd* pointer can be manipulated, a fake chunk can be crafted. For the fake chunk, the *size* value has the least significant bit set to 0 (*PREV\_INUSE* off). The *prev\_size* and *size* value has to be small enough. Large values may return an in memory access error. It also has to be noted that 12 has to be subtracted from the address which we are trying to overwrite.

<b>prev_size = 0xffffffffc</b>
<b>size = 0xffffffffc</b>
<b>fd = return location - 12</b>
<b>bk = return address</b>

Figure 35.0 showing fake chunk

The address of the *return location -12* will overwrite *bk +8*. A jump instruction has to be inserted at the *return address*. This will get past the malicious instruction at the *return address + 8*. Figure 36.0 shows the two chunks, after the overflow occurs with the chunk. The overwritten chunk is unlinked when the second free in vulnerable program is called. For the exploit code to get executed, the pointer *bk* should point to it.

<b>prev_size = unknown</b>
<b>size = 1024</b>
<b>data(p1) = padding</b>
<b>prev_size = 0xffffffffc</b>
<b>size = 0xffffffffc</b>
<b>fd = return location - 12</b>
<b>bk = return address</b>

Figure 36.0 showing over written chunk

The vulnerability exists in MJPEG codec routine. The reason of the vulnerability is inappropriate validity that results in the mismatch between the buffer allocation size and the number of bytes to be read.

### AVI File format:

Figure 36.1 is the structure diagram describing the RIFF from of the MPEG AVI format.

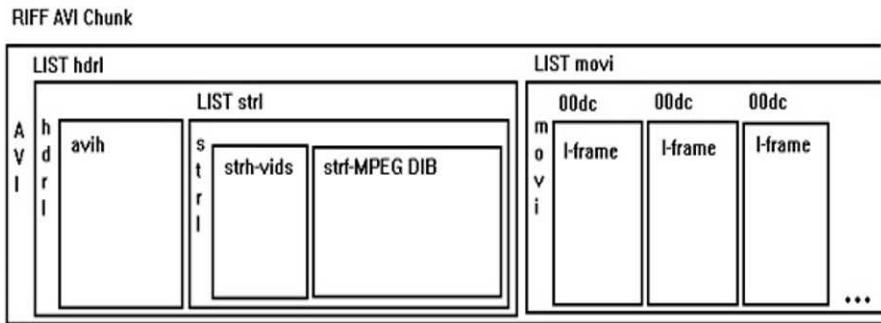


Figure 36.1 showing the RIFF diagram from MPEG AVI format.

### Brief Technical Details

When MJPEG AVI format file is encountered in DirectX software, AVI filter communicates with the MJPEG decoder filter. At the time of establishing the communication with MJPEG decoder, information about the dimension of the image is sent. This dimension information is required at the initialization because output sample buffer is allocated accordingly. As AVI filter should know the information about the dimension of the image the dimension information is indicated in a BITMAPINFOHEADER structure. This BITMAPINFOHEADER structure is found at the stream format chunk. *So the memory allocation for streaming is based on the dimension information present in the BITMAPINFOHEADER structure.*

MJPEG data format has internal header that indicates the dimension information about the image. MJPEG decoder parses the dimension information stored in the header and accordingly determines the number of bytes to be read in the allocated buffer. *Its mandate to ensure image dimensions information stored in BITMAPINFOHEADER and MJPEG internal header should match.*

### Manipulating AVI file

To exploit this vulnerability, the malicious AVI file should indicate the dimension (size) information of image stored in the *BITMAPINFOHEADER* less than the dimension information indicated by the header of MJPEG file. This will make number of byte to read greater than allocation size which in turn results in heap overflow.

This can be achieved either by decreasing the dimension indicated in the image size tag in *BITMAPINFOHEADER structure* or by increasing the size indicated in the MJPEG structure.

### Approach 1:

Decreasing the number indicated in the image size tag in *BITMAPINFOHEADER* structure Use “010” editor to manipulate the value *BITMAPINFOHEADER* as indicated below

	0 1 2 3 4 5 6 7 8 9 A B C D E F	0123456789ABCDEF
0010h:	94 FC 00 00 68 64 72 6C 61 76 69 68 38 00 00 00	“u..hdrlavih8...
0020h:	57 82 00 00 B4 1F 04 00 01 00 00 00 10 08 00 00 00	W...’.....
0030h:	95 04 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00	*.....*
0040h:	00 00 00 00 78 00 00 00 00 00 00 00 00 00 00 00 00 00	....x.....
0050h:	00 00 00 00 00 00 00 00 00 00 00 4C 49 53 54 AC 7D 00 00	.....LIST-}..
0060h:	73 74 72 6C 73 74 72 68 38 00 00 00 76 69 64 73	strlsth8...vids
0070h:	4D 4A 50 47 00 00 00 00 00 00 00 00 00 00 00 00 00 00	MJPG.....
0080h:	66 17 05 00 80 96 98 00 00 00 00 00 96 04 00 00	f...€-”.....
0090h:	B0 12 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	“.....
00A0h:	B4 00 78 00 73 74 72 66 40 00 00 40 00 00 00 00 00 00	‘.x.strf0...@...
00B0h:	B4 00 00 00 78 00 00 01 01 18 00 4D 4A 50 47	‘.x.....MJPG
00C0h:	40 FA 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	01. ....
00D0h:	00 00 00 00 18 00 00 00 00 00 00 00 00 02 00 00 00 00	.....
00E0h:	08 00 00 00 02 00 00 02 00 00 00 69 6E 64 78	.....indx

Name	Value	Start	Size	Color
DWORD xdwQuality	0	9Ch	4h	Fg: Bg:
DWORD xdwSampleSize	7864500	A0h	4h	Fg: Bg:
struct strFHEDER_BIH strf		A4h	48h	Fg: Bg:
char id[4]	strf	A4h	4h	Fg: Bg:
uint32 datalen	64	A8h	4h	Fg: Bg:
struct BITMAPINFOHEADER bmiHeader		ACh	28h	Fg: Bg:
uint32 biSize	64	ACh	4h	Fg: Bg:
uint32 biWidth	180	B0h	4h	Fg: Bg:
uint32 biHeight	120	B4h	4h	Fg: Bg:
uint16 biPlanes	1	B8h	2h	Fg: Bg:
uint16 biBitCount	24	BAh	2h	Fg: Bg:
uint32 biCompression	1196444237	BCh	4h	Fg: Bg:
uint32 biSizeImage	1129600	C0h	4h	Fg: Bg:
uint32 biXPelsPerMeter	0	C4h	4h	Fg: Bg:
uint32 biYPelsPerMeter	0	C8h	4h	Fg: Bg:
uint32 biClrUsed	0	CCh	4h	Fg: Bg:
uint32 biClrImportant	0	D0h	4h	Fg: Bg:
char exData[24]	□	D4h	18h	Fg: Bg:
struct strFHEDER strn		ECh	7D20h	Fg: Bg:
struct LISTHEADER list[2]		7E0Ch	70h	Fg: Bg:

Figure 36.2 Snapshot of file using 010 editor using original value

### Approach 2:

Increasing the size indicated in the following MJEPAGE header structure. This header structure is preceded with each I-frame( in steaming sub chuck).

Header code (4 bytes)  
width (4 bytes, integer, MSB)  
height (4 bytes, integer, MSB)  
aspectRatioCode (4 bytes, integer, MSB)  
frameRateCode (4 bytes, integer, MSB)  
bitRate (4 bytes, integer, MSB)  
vbvBufferSize (4 bytes, integer, MSB)  
constrainedParamFlag (4 bytes, integer, MSB)

By increasing the dimension of the image in the above data structure will result into the heap overflow.

**Note:** - we cannot write exploit by only using approach 1

Address	Value	Start	Size	Color
0010h	94 FC 00 00 68 64 72 6C 61 76 69 68 38 00 00 00			
0020h	57 62 00 00 B4 1F 04 00 01 00 00 00 10 08 00 00			
0030h	95 04 00 00 00 00 00 00 02 00 00 00 80 12 00 00			
0040h	00 00 00 00 78 00 00 00 00 00 00 00 00 00 00 00			
0050h	00 00 00 00 00 00 00 00 4C 49 53 54 AC 7D 00 00			
0060h	73 74 72 6C 73 74 72 68 38 00 00 00 76 69 64 73			
0070h	4D 4A 50 47 00 00 00 00 00 00 00 00 00 00 00 00			
0080h	66 17 05 00 80 96 98 00 00 00 00 96 04 00 00			
0090h	80 12 00 00 00 00 00 00 00 00 00 00 00 00 00 00			
00A0h	B4 00 78 00 73 74 72 66 40 00 00 40 00 00 00 00			
00B0h	50 00 00 00 78 00 00 00 01 00 18 00 4D 4A 50 47			
00C0h	A0 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00			
00D0h	00 00 00 00 18 00 00 00 00 00 00 00 02 00 00 00			
00E0h	08 00 00 00 02 00 00 00 02 00 00 00 69 6E 64 78			

Name	Type	Value	Start	Size	Color
DWORD xdwQuality	DWORD	0	9Ch	4h	Fg: Bg:
DWORD xdwSampleSize	DWORD	7864500	A0h	4h	Fg: Bg:
struct strfHEADER_BIH strf	struct strfHEADER_BIH strf		A4h	48h	Fg: Bg:
char id[4]	char id[4]	strf	A4h	4h	Fg: Bg:
uint32 datalen	uint32	64	A8h	4h	Fg: Bg:
struct BITMAPINFOHEADER bmiHeader	struct BITMAPINFOHEADER bmiHeader		ACh	28h	Fg: Bg:
uint32 biSize	uint32	64	ACh	4h	Fg: Bg:
uint32 biWidth	uint32	80	B0h	4h	Fg: Bg:
uint32 biHeight	uint32	120	B4h	4h	Fg: Bg:
uint16 biPlanes	uint16	1	B8h	2h	Fg: Bg:
uint16 biBitCount	uint16	24	BAh	2h	Fg: Bg:
uint32 biCompression	uint32	1196444237	BCh	4h	Fg: Bg:
uint32 biSizeImage	uint32	12960	C0h	4h	Fg: Bg:
uint32 biPelsPerMeter	uint32	0	C4h	4h	Fg: Bg:
uint32 biYelsPerMeter	uint32	0	C8h	4h	Fg: Bg:
uint32 biClrUsed	uint32	0	CCh	4h	Fg: Bg:
uint32 biClrImportant	uint32	0	D0h	4h	Fg: Bg:
char exData[24]	char exData[24]		D4h	18h	Fg: Bg:
struct strnHEADER strn	struct strnHEADER strn		ECh	7D20h	Fg: Bg:

Figure 36.3 showing manipulated value

## Reversing to analyze Vulnerability

The fix for this vulnerability is available with the patch “**KB951698**”. The name of the vulnerable binary is **quartz.dll**. Apply bindiff on vulnerable binary with the patched binary (quarts.dll). Figure 36.4 is the output of the bindiff showing the list of the functions that are changed.

	Vulnerable binary	Patched binary
res	748293f6 sub_748293F6	7489bf5a sub_7489BF5A
res	74838903 sub_74838903	74838636 sub_74838636
res	7484b6eb sub_7484B6EB	7484b6a9 sub_7484B6A9
res	7487e866 sub_7487e866	7487e5ee sub_7487E5EE
res	74883c0f sub_74883C0F	74883d9f sub_74883D9F
res	7488442c sub_7488442C	748843be sub_748843BE
res	748844c2 sub_748844C2	74884447 sub_74884447
res	748845f8 sub_748845F8	7488457a sub_7488457A
res	7488547f sub_7488547F	74887569 sub_74887569
res	74886242 sub_74886242	74884ec4 sub_74884EC4
res	748864d7 sub_748864D7	748872cd sub_748872CD
res	74886927 sub_74886927	7488c13d sub_7488C13D
res	7488783a sub_7488783A	7488771d sub_7488771D
res	74887fc0 sub_74887FC0	74889f19 sub_74889F19
res	74888446 sub_74888446	748a01a5 sub_748A01A6
res	74888ae0 sub_74888AE0	74888933 sub_74888933
res	748893da sub_748893DA	7488926a sub_7488926A
res	7488ad0e sub_7488AD0E	7488ab68 sub_7488AB68
res	7488b37c sub_7488B37C	7488677e sub_7488677E
res	7488c396 sub_7488C396	7488e100 sub_7488E100
res	7488cb5e sub_7488CB5E	7488c926 sub_7488C926
res	7488df2 sub_7488DF2	7488cebo sub_7488CEBO
res	7488e228 sub_7488E228	7488f0af sub_7488FOAF
res	7488e8e3 sub_7488E8E3	7488e8a5 sub_7488E8A5
res	748a207a sub_748A207A	748965e4 sub_748965E4
res	748ac8e0 sub_748AC8E0	748777d54 sub_74877D54
res	748bc714 sub_748BC714	748bc8c1 sub_748BC8CF
res	748c196a sub_748C196A	748c1a92 sub_748C1A92
res	748c1c91 sub_748C1C91	748c1d85 sub_748C1D85
res	748c1lee5 sub_748C1LEE5	748c1fd7 sub_748C1FD7
res	748c3053 sub_748C3053	748c315c sub_748C315C
res	748c3301 sub_748C3301	748c3379 sub_748C33F9
res	748c532d sub_748C532D	748c54c5 sub_748C54C5
res	748c5449 sub_748C5449	748c55d9 sub_748C55D9
res	748c5bda sub_748C5BDA	748c5d3a sub_748C5D3A
res	748c6ddf sub_748C6DDF	748c6e40 sub_748C6E40
res	748c6f43 sub_748C6F43	748c6fae sub_748C6FAE
res	748d8dd0 sub_748D8DD0	748d8d13 sub_748D8D13
res	748dc9a4 sub_748DC9A4	748dc85c sub_748DC85C
res	748fb8d7 sub_748FB8D7	7490417 sub_74904177
res	748ff410 sub_748FF410	748780b6 sub_748780B6
res	74901e36 sub_74901E36	748bf795 sub_748BF795
res	7490a4ef sub_7490A4EF	7490a3a3 sub_7490A3A3
res	7490e3c6 sub_7490E3C6	7490e1dc sub_7490E1DC
res	748d8ad2 sub_748D8AD2	748d8b9b sub_748D8B9B

Figure 36.4 showing the output of bindiff function. .

There are approximately 50 functions that have been changed for the patch binary.

Vulnerable binary	Patched Binary	ID
7484b6eb sub_7484B6EB	7484b6a9 sub_7484B6A9	1
7488442c sub_7488442C	748843be sub_748843BE	2
748844c2 sub_748844C2	74884447 sub_74884447	3
748845f8 sub_748845F8	7488457a sub_7488457A	4
74886242 sub_74886242	74884ec4 sub_74884EC4	5
748864d7 sub_748864D7	748872cd sub_748872CD	6
7488783a sub_7488783A	7488771d sub_7488771D	7
7488ad0e sub_7488AD0E	7488ab68 sub_7488AB68	8
7488b37c sub_7488B37C	7488677e sub_7488677E	9
7488c396 sub_7488C396	7488e100 sub_7488E100	10

Figure 36.5 showing functions which have been modified.

Carefully analyzing the function sub\_7488AB68 (ID 8) will take close to the routine where the fixed code has been added. The address of the routine in which validation has been added **74882EBA** and its equivalent subroutine in vulnerable dll is at **74882EFC**

### Vulnerability 1: -

The assembly code shown in figure 36.6 is vulnerable because the size of image (number of bytes to be read) is calculated using the parameter present in JPEG header and the JPEG header parameters are not validated against the parameters stored in the BITMAPINFOHEADER that is used for calculating the size for allocating memory.

```

push  esi
call sub_74884AC2
mov al, [edi+300h]
test al, al
jnz short loc_74882FEA
mov eax, [ebx+8Ch]
mov esi, [eax+4] ; Reading biWidth data member from structure BITMAPINFOHEADER
imul esi, [edi+224h] ; Accessing JPEG header information to calculate the number bytes for each scan line
add esi, 3
and esi, 0xFFFFFFFFCh
mov [ebp+var_2024], esi
mov eax, [eax+8] ; Reading biHeight data member from structure BITMAPINFOHEADER
dec eax

loc_74882FDC:
    imul eax, esi ; CODE XREF: sub_74882EFC+120j
    add eax, [ebx+10h] ; Adding the calculated size to the allocated stream buffer

loc_74882FE2:
    mov [ebp+var_2020], eax ; Filling of the stream buffer will be done from the end
    jnp short loc_74883062
;

```

Figure 36.6 showing the assembly

### Vulnerability 2: -

Figure 36.7 shows the code for data into the streaming buffer. The filling of buffer is done in reverse order. If the pointers into the allocated buffer decrement below the beginning of the allocated streaming buffer the rest of the scan lines are read into a stack buffer of size 0x2000. The code shown in figure 36.7 is vulnerable as the scan line can be of any size and the size of the buffer is 0x2000. This can result into buffer overflow.

```

.text:74883050
.text:74883056 loc_74883056:           ; CODE XREF: sub_74882EFC+124↑j
    mov    esi, [ebp+var_2024]
.text:74883056
.text:7488305C loc_7488305C:           ; CODE XREF: sub_74882EFC+1AC↓j
    mov    eax, [ebp+var_2020]
.text:7488305C
.text:74883062
.text:74883062 loc_74883062:           ; CODE XREF: sub_74882EFC+EC↑j
    mov    eax, [edi+290h] ; Number of time loop executed
    cmp    ecx, [edi+27Bh] ; Accessing JPEG header to get height of the image
    jnb    short loc_748830AA ; Exit if loop count is greater than image size
    push   1
    push   eax
    cmp    eax, [ebx+10h] ; Compare if the offset pointer has gone down
    jbe    short loc_74883096 ; If the offset pointer gone down plz copy it in buffer
    lea    eax, [ebp+var_2020]
    push   eax
    push   [ebp+var_2020] ; Storing in Heap
    call   sub_74884593
    mov    eax, [ebp+var_2020]
    sub    eax, esi
    jmp    loc_74882FE2
.text:74883096
.text:74883096 loc_74883096:           ; CODE XREF: sub_74882EFC+179↑j
    lea    eax, [ebp+var_2034]
    push   eax
    push   [ebp+var_2028] ; storing in buffer
    push   call sub_74884593
    jmp    short loc_7488305C
.text:748830A8
.text:748830AA ; -----
retf 748830AA

```

Figure 36.7 shows the assembly code

### Writing Exploit:-

The binary can be exploited in various ways. Following are the two techniques that can be used to write an exploit

#### Technique 1:-

1. Change the width value stored in the JPEG header such that it becomes greater than the “biwidth” value stored in *BITMAPINFOHEADER* structure.
2. This will result in heap overflow.

Following is the reason

The following two lines of the “vulnerable code 1 snippet” calculates the number of bytes for each scan line

```

.text:74882FC2      mov    esi, [eax+4] ;
.text:74882FC5      imul   esi, [edi+224h] ;

```

Here the first line access “biwidth” value stored in *BITMAPINFOHEADER* structure and the second statement is access the JPEG header information to calculate the number bytes for each scan line. If we increase this value in the JPEG header heap overflow can very easily be triggered. That can

result in overwriting the next pointer value of the heap data structure.

### Technique 2:-

1. Change the value of “biheight” member of *BITMAPINFOHEADER* structure such that the image height stored in the JPEG header should be greater than the *BITMAPINFOHEADER.biheight*. This will force the binary to fill scan lines in the buffer which is of size 2000.

Following is the reason for it

Reading size is calculated by referring the *BITMAPINFOHEADER.biheight*. Please refer following line from the “vulnerable code 1 snippet”

```
.text:74882FD8      mov    eax, [eax+8]
.text:74882FDB      dec    eax
.text:74882FDC      imul   eax, esi
```

Here the first line access “biheight” value stored in *BITMAPINFOHEADER* structure.

Number of time loop need to be executed is determined by the value stored as a height field in the JPEG header.

Please refer following line from the “vulnerable code 2 snippet”

```
.text:74883068      cmp    ecx, [edi+278h]
.text:7488306E      jnb    short loc_748830AA
```

So buffer overflow can be triggered if we change the JPEG header accordingly and rest of the scan lines will be stored in the buffer of limited size.

## 4.4 Integer Overflows

Integer overflow happens because of the values held in a numeric data types are limited by the data type’s size in bytes. In ANSI C, the character’s size is 1 Byte, short takes 2 bytes, integer takes 2 bytes and long is 4 bytes in size. The range of data types depends upon whether they are signed or unsigned. A signed 2 bytes short will take values from -32767 and 32767 , whereas an unsigned short will take values between 0 and 65535. This is important to note because if an attempt is made to put a value in a data type which is too small to hold for unsigned data types, the higher order bits will be dropped. Modulo arithmetic is performed on the value before they are stored.

Stored = value % (limit +1)

This ensures that high unsigned values fit the data type. Explaining it with an example, if the maximum value which an unsigned data type can hold is 65535 and if a value of 65536 is entered, then 0 will be stored. This can be computed by  $65536 \% (62535 + 1) = 0$

---

```

include <string.h>
include <stdio.h>

nt main() {
    unsigned int a = 10;
    unsigned int b = 20;
    int c = 100;
    int d = 200;

    if(a>b)
        printf ("Unsigned Integer Comparison\n");

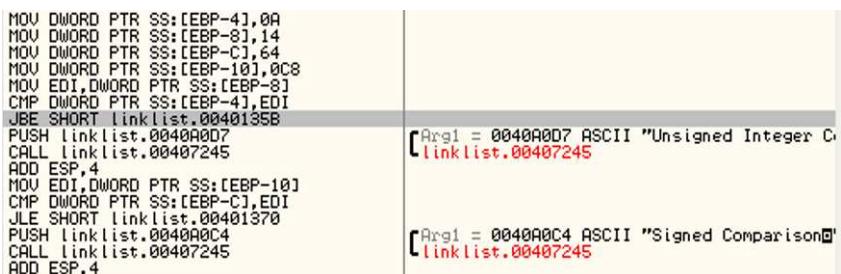
    if(c>d)
        printf ("Signed Comparison\n");

}

```

Figure 19.0 C code showing signed and unsigned comparison

For signed data types, the first half of the range (0 thru 0111 1111 1111 1111) are used for positive numbers. This is done in order of least to greatest. The second half of the range is used for negative numbers in order of least to greatest. So if the maximum limit for the signed type is 32767, and if 32768 is being stored in the data type, it will be represented as 1000 0000 0000 0000 and the value -32768 will be stored.



The screenshot shows assembly code for the C code in Figure 19.0. The assembly code includes instructions for both unsigned and signed comparisons. The assembly code is as follows:

```

MOV DWORD PTR SS:[EBP-4],0A
MOV DWORD PTR SS:[EBP-8],14
MOV DWORD PTR SS:[EBP-C],64
MOV DWORD PTR SS:[EBP-10],0C8
MOV EDI,DWORD PTR SS:[EBP-8]
CMP DWORD PTR SS:[EBP-4],EDI
JBE SHORT linklist.0040135B
PUSH linklist.0040A0D7
CALL linklist.00407245
ADD ESP,4
MOV EDI,DWORD PTR SS:[EBP-10]
CMP DWORD PTR SS:[EBP-C],EDI
JLE SHORT linklist.00401370
PUSH linklist.0040A0C4
CALL linklist.00407245
ADD ESP,4

```

Annotations on the right side of the assembly code highlight specific memory addresses and their values:

- Arg1 = 0040A0D7 ASCII "Unsigned Integer C**  
linklist.00407245
- Arg1 = 0040A0C4 ASCII "Signed Comparison**  
linklist.00407245

Figure 20.0 showing the assembly instruction for signed and unsigned comparison in C code shown in figure 19.0

Conditional and unconditional codes are the two different conditional codes in IA-32 assembly language. The conditional code used in a conditional jump exposes the exact data type, used in comparison, in the original source code. As shown in figure 20.0, if the instruction JG /JLE/JGE is used then it denotes that the buffer length/length is treated as a signed integer by the compiler. If the instruction JA/JBE/JAE/JNB is used, then the compiler is treating buffer length/length as the unsigned integer.

#### 4.4.1 Types Integer Overflow

Signed overflow bugs happen generally when

- Signed integers are used for comparison
- Signed integers are used for arithmetic
- Unsigned integers are compared with the signed integer.

---

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
    int b = 0x7fffffff;
    char buffer[20];

    printf("The value of b + 1 is %d \n", b+1);

    if( (b +1 ) < 20) {
        strcpy(buffer,"I am overflowing buffer");
    }
}
```

Figure 21.0 c code for signed integer comparison leading to overflow.

Signed integer overflow is explained with the example shown in figure 21.0. As shown in figure 21.0, the signed integer b has value 0x7fffffff. Character buffer of size 20 is allocated. As shown in figure 22.0, in the assembly, this value is loaded in the DWORD PTR SS:[EBP-4]. This value is then moved to EDI, which is incremented by 1. This is then compared with 20. Since the instruction JGE is used in comparison, from assembly it can be inferred that the value 0x7fffffff stored in ESI, has been declared as signed. So the instruction EDI +1, will result in -2147483648. This is less than 0x14 so the comparison will be false, resulting in a buffer overflow. Here, it can be

seen that signed integer comparison can result in passing the checks. In many of the applications, these types of checks are present to ensure that the value passed inside the buffer is less than the size of the buffer. Buffer length should always be an unsigned value as there cannot be negative or zero buffer length. Hence signed buffer checks which can be identified by instructions JG/JGE/JLE can result in an overflow.

<pre>     .C745 FC FFFFFF MOV DWORD PTR SS:[EBP-4],?FFFFFFF     .8B7D FC MOV EDI,DWORD PTR SS:[EBP-4]     .83C7 01 ADD EDI,1     .57 PUSH EDI     .68 B9A0400000 PUSH testabh.0040A0B9     .E8 F75E0000 CALL testabh.004071F9     .83C4 08 ADD ESP,8     .8B7D FC MOV EDI,DWORD PTR SS:[EBP-4]     .83C7 01 ADD EDI,1     .83FF 14 CMP EDI,14     .7D 11 JGE SHORT testabh.00401321     .68 A0A0400000 PUSH testabh.0040A0A0     .8D7D E8 LEA EDI,DWORD PTR SS:[EBP-18]     .57 PUSH EDI     .E8 072E0000 CALL testabh.00404125     .83C4 08 ADD ESP,8     &gt; B8 00000000 MOV EHX,0     .5F POP EDI   </pre>	<p>Arg2 Arg1 = 0040A0B9 ASCII "The value of b + <b>testabh.004071F9</b></p> <p>ASCII "I am overflowing buffer@"</p>
--	---

Figure 22.0 showing the assembly code for the C code shown in figure 7.0

For signed integer overflow, movsx is the other instruction which should be monitored in assembly code for overflow.

movsx esi, DWORD PTR SS:[EBP-4]

Figure 23.0 copying using movsx instructions

The code shows in Figure 23.0 copies the parameters from stack to esi register. It treats the length stored in SS:[EBP-4] as signed short and hence the sign extends it. In case, the parameter (which could be buffer length) has its most significant bit set, it will be converted into a negative 32-bit number. For example, if the length is 0x9600 (equivalent to 38400 in decimal), it will become 0xffff9600 (equivalent to 4294940160). If the esi holds the length of data which can be copied in the buffer, the instruction shown in figure 23.0 will result in buffer overflow. If the length is defined as an unsigned short, the instruction MOVZX will be used. MOVZX makes the extended integer zero during conversion. The most significant word in the target 32-bit integer is set to zero. Hence the numeric value remains the same.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main() {
    unsigned int b = 0xffffffff;
    char *buffer;

    printf("The value of b + 1 is %d \n", b+1);

    if( (b +1 ) < 20) {

        buffer = (char*)malloc(b+1);
        strcpy(buffer, " I am overflowing buffer");

    }
}
```

Figure 24.0 C code for the Unsigned Integer overflow.

Like the signed integer overflow, unsigned integers can also be used to perform overflow. As discussed earlier for an unsigned integer, if an attempt is made to store the value which is greater than the data type, the integer will be truncated. As shown in figure 24.0 for the C code, the value of the unsigned int b is 0xffffffff. The operation b+1, will result in 0, hence the buffer will be allocated and will be overflowed by strcmp instruction. As shown in figure 25.0, in the assembly, the instruction JNB is used. JNB denotes that the value stored in EDI is an unsigned integer.

```
PUSH EBP
MOV EBP,ESP
PUSH ECX
PUSH EAX
MOV ECX,2
DEC ECX
MOV DWORD PTR SS:[ESP+ECX*4],FFFA5ASA
JNB SHORT testvr.004012DE
PUSH EDI
MOV DWORD PTR SS:[EBP-4],-1
MOV EDI,DWORD PTR SS:[EBP-4]
ADD EDI,1
PUSH EDI
PUSH testvr.004000A0
CALL testvr.004071F9
ADD ESP,8
MOV EDI,DWORD PTR SS:[EBP-4]
ADD EDI,1
CMP EDI,14
JNB SHORT testvr.00401321
MOV EDI,DWORD PTR SS:[EBP-4]
ADD EDI,1
PUSH EDI
CALL <JMP.&CRTDLL.malloc>
ADD ESP,4
MOV DWORD PTR SS:[EBP-8],EAX
MOV EAX,0
POP EDI
LEAVE
RETN
```

Figure 25.0 Assembly code for the C code shown in figure 9.0 showing unsigned Integer Overflow.

In an application code, the integer overflow can be exploited, when calculation is made, about how large a buffer must be allocated. Application code generally makes use of calloc or malloc routines. These routines aid the reservation of space by multiplying the number of elements by the size of an object. Signed/unsigned checks can lead to error conditions. If there are some comparison checks signed bufferlen <= Max buffer size, or if the signed buffer length is less than zero, it will satisfy the condition that the signed buffer length should be less than the maximum buffer size. It also has to be noted that the buffer lengths which are entered as inputs cannot be negative values. A negative value is treated as a very large number. If the memory allocation has been done based on an unsigned integer data type or if the value is wrapped around then there will be very less memory. If the comparison is done based on the signed integer value, and some other number, or the condition signed integer value is less than the other number value, and if the other number value has overflowed into a negative value, the comparison will pass.

#### 4.4.2 CAN-2004-0417 CVS Max dotdot Protocol Command Integer Overflow

A new memory buffer is allocated when the CVS server gets a Max-dotdot command and converts it into a numeric string. This numeric string is passed by the CVS client to an integer value. For storing the path names, the CVS server allocates the memory which is two times larger than the supplied number in the Max-dotdot command along with the length of original temporary path names. The C code shown in figure 26.0 shows the vulnerable function in the server.c file which performs the calculation.

---

```
serve_max_dotdot (arg)
    char *arg;
    int lim = atoi (arg);
[...]
    (lim < 0)
    return;
    p = xmalloc (strlen (server_temp_dir) + 2 * lim + 10);
```

Figure 26.0 Vulnerable C code in Server.c

It can be seen that the lim is defined as a signed integer. 2147483634 is the minimum value for a 32 bit platform which will result in integer overflow. The code that fills the new buffer is shown in figure 27.0

```

strcpy (p, server_temp_dir);
for (i = 0; i < lim; ++i)
    strcat (p, "/d");

```

Figure 27.0 C code filling the buffer

%d	It converts an integer to a signed decimal string
%u	This converts an integer to an unsigned decimal string
%i	This converts an integer to a signed decimal string. An integer may be in decimal or octal format.
%o	This converts the integer to unsigned octal strings
%X	This converts an integer to unsigned hexadecimal string
%c	This converts an integer to the Unicode characters it represents
%s	This inserts the string
%f	This converts the floating point number to a signed decimal string
%e or %E	This converts the floating point to a scientific notation in the form x.yyye+zz. If the precision is 0, then no decimal point is displayed in the output.
%g or %G	Uses exponential format if exponent is greater than -4 or less than precision, decimal format otherwise.
%n	Records the number of characters so far.
%r	String (converts any python object using <code>repr()</code> ).
%p	The <code>void *</code> pointer argument is printed in hexadecimal

Figure 28.0 Format Specifiers in C function

In case of integer overflow, the memory buffer will be too small to store all the data copied to it, hence a heap overflow will be triggered.

## 4.5 Format String

Various functions like `printf()`, `fprintf()`, `vprintf()` and `sprintf()` use formats strings. The format gives the programmer a degree of control over

how the text should be printed, therefore allowing the programmer to control the output. Figure 28.0 shows the list of format specifiers in the C function. These format functions take the format strings as the first argument and an equal number of variables for the format strings. Therefore, if four format specifiers exist in a function there will be four arguments in the function. The format string controls the behavior of the format function. The function retrieves the parameters requested by the format string from the stack. For example for the C instruction

```
printf("The value of %d : %08x\n", a, &a);
```

from within the printf function the stack looks like:

ESP → Return Address

ESP+4 → Offset of string "The value of %d : %08x\n"

ESP+8 → Value of a

ESP+12 → Address of a

The format function now parses the format string 'a', by reading a single character at a time. If it is not '%', the character is copied to the output. If the character is %, the character behind the '%' specifies the type of parameter that should be evaluated. The string '\%' has a special meaning: it is used to print the escape character '%'. Every other parameter relates to data, which is located on the stack. The format specifiers direct the function to read from the corresponding arguments. If the address is not in the valid range it might result in a read violation

#### 4.5.1 Format String Vulnerability

The behavior of the function can be controlled by using format strings. Poorly written C programs use `printf(string1)` (lets call it a first function), instead of `printf("%s",string1)` (Lets call it a second function). Functionally, the first function works well. The format function is passed to the address of the string, as compared to the address of a format string and it iterates the printing of each character.

---

```
#include<stdio.h>
#include<string.h>

void main(){
    char buffer[100];
    strcpy(buffer, "Abhishek");
    printf(buffer);
    printf("%s", buffer);
}
```

Figure 29.0 showing C code with and without format specifiers.

However, if String string1 = “%08x.%08x.%08x.%08x” in the function printf(string) and is passed as a parameter then, the printf function will print the address of memory locations instead of the value of the string. This is exploited for format string vulnerability. The functions that are prone to format string vulnerabilities are printf, fprintf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf.

```
call  strcpy
add   esp, 8
lea   ecx, [ebp+var_64]
push  ecx
call  printf
add   esp, 4
lea   edx, [ebp+var_64]
push  edx
push  offset asc_42001C ; "%"
call  printf
```

Figure 30.0 showing the assembly code by IDA pro for the C code in figure

The figure 29.0 shows the C code with and without format specifiers. The first printf( ) does not makes use of format specifiers and is prone to format string vulnerability. In the case of the second printf(), a format specifier is specified. Figure 30.0 shows the assembly instructions by using IDA pro disassembler of the C code shown in figure 29.0. For the printf without format specifiers, as seen in assembly, only the variable is pushed on to the stack., where as for the second printf, format specifiers along with the variable are pushed onto the

stack. While analyzing binaries which are not prone to format string vulnerability, if n number of arguments is pushed on to the stack before making a call to the printf family of functions, an n number of format specifiers should be pushed onto the stack. If the number of format specifiers is less than number of arguments, then the assembly code is prone to format string vulnerability.

The attack due to the format string vulnerability can be divided into three parts: format string vulnerability denial of service attack; format string vulnerability reading attack and format string vulnerability writing attack. The format specifier “%n”, directs the function to store the number of characters that have been output so far to an integer indicated by a pointer to an argument. This conversion specifier gives the attacker a capability to write to the random memory address and perform format string write attacks.

#### 4.5.2 Format String Denial of Service Attack

The format strings vulnerabilities can be used to make a process crash. In UNIX, illegal pointer access is caught by a kernel and it sends a SIGSEGV signal. The process is terminated and dumps core. Supplying format strings can easily trigger invalid pointer accesses and hence perform a denial of service attack.

```
printf( "%n%n%n%n%n%n%n\n" );
```

Figure 31.0 printf functions with format specifiers

In figure 31.0, %d will display memory from an address that is supplied on the stack, which stores other data also. If a large number of %d are specified, then an instruction might read from illegal addresses, which are not mapped. This in turn will result in a denial of service attack. Similarly, %s can also be used to read the data from the stack. Again, a large number of %s will try to read the data from illegal addresses, which again will result in a crash.

#### 4.5.3 Format String Vulnerability Reading Attack

Format strings can be used to perform reading attacks where the content of stacks can be viewed. For example, C instructions like

printf("%08x.%08x.%08x.%08x\n"); will give the following output:

0012ffc0.0040212bc.00000001.00144d28.00144440

This is a partial dump of the stack memory. Based on the size of the format string and the size of the output buffer, a large part of stack memory can be reconstructed. It is also possible to retrieve the entire stack memory. The %s format parameter can be used to read from the memory address. The %s can retrieve the address and print the desired value. If, in the C instruction the fourth parameter is %s,

```
printf("%08x.%08x.%08x.%s\n");
```

the value located at the address 0x00144440 will be printed. If the value at the address is a string or the address is of a legal value, then the value will be printed. This information can in turn be used to find out the flow of program, local variables and can be used for successful exploitation. If the value of address is not a legal value, as seen earlier, it will result in a segmentation fault. %x, %d and %c are the format specifiers which can be used to view the content of stacks. %x and %d retrieve the double word from the stack and display them in hexadecimal or decimal format. The format specifier %x displays only one double word, which is located on the top of the stack. Format specifier %c, retrieves the paired double word from the stack. It then converts it into the single byte of type character and displays it as a character, discarding the three most significant bytes. Hence, N specifiers display  $4^*N$  bytes. The maximum depth is equal to  $2^*Y$ , where Y is the maximum allowed size of user input in bytes.

## 4.6 SEH (Structure Exception Handler)

SEH (Structure Exception Handler) is one of the most reliable ways to gain the code execution flow to execute shell code through stack-based overflow. SEH structure exception handler mechanism can be used to handle both hardware as well as software exceptions. Through SEH, the application can dynamically register and unregister exception handler function in SEH chain. And at the time of exception, this SEH chain is accessed and each of the exception handlers in the chain is given the opportunity to either handle the exception or pass it on to the next exception handler. The way the chain is maintained and accessed is discussed below. The sample asm program shown in figure 38.1 explains it in detail.

---

```
#include <windows.h>

void main(int argc, char* argv[])
{
    int i = 6;
    int j;
    {
        _try
        {
            i = i + 3;
            j = i % 9;
            i = i / j;
        }
        _except(EXCEPTION_EXECUTE_HANDLER)
        {
            printf("Exception Zero");
        }
    }
}
```

Figure 38.1 C code having exception

The list of registered exception handler functions is maintained through linked list data structure. This list is called SEH chain. The nodes of these lists are created on the stack.

The structure of each node of this link list is as follows:-

```
_EXCEPTION_REGISTRATION {
    struc prev dd ?
    handler dd ?
}_EXCEPTION_REGISTRATION ends
```

The operating system maintains this list for each and every thread.

The disassembled code shown in figure 38.2 of application in figure 38.1 will explain in great detail about the internals how SEH blocks are registered and maintained in the chain. The instruction in figure 38.2 registers the new exception handler in the SEH chain.

```

    mov    ebp, esp
    push   0xFFFFFFFF
    push   offset unk_4050A8
    push   offset unknown_libname_1
    mov    eax, large fs:0
    push   eax
    mov    large fs:0, esp
    sub    esp, 10h
    push   ebx
    push   esi
    push   edi
    mov    [ebp+var_18], esp
    xor    eax, eax
    mov    [ebp+var_4], eax
    mov    [ebp+var_1C], 9
    mov    [ebp+var_20], eax
    mov    eax, 9
    cdq
    xor    ecx, ecx
    idiv  ecx
    mov    [ebp+var_1C], eax
    jmp    short loc_40105A

    mov    eax, 1
    ret

```

Figure 38.2 showing the disassembled code of C

The first instruction PUSH XXXX. Push is the address of the exception handler function. This is the function that needs to be called when an exception occurs while executing instruction the current thread context. After pushing the exception handler function a next instruction is to Push the fs[0] value. Fs registers can be used to access the thread environment block of currently executing thread. And the very first four byte of the thread environment block points to the head of the SEH chain. The instruction push fs[0] pushes the start of the SEH chain address in Stack. Pushing these two values creates a complete exception node in the stack. And the current value of ESP points to this newly created node. The next instruction mov fs[0] , esp moves the address of the newly created node into the TEB of currently executing thread. This implies three instructions are creating a node and adding an exception handler node in the SEH chain in which the first two instruction is creating a node in the stack and the last instruction adds the newly created node at the top of the list. When an exception occurs in any thread this chain in accessed and exception handlers listed in the top of the node is invoked first.

The SHE chain can be viewed in olly dbg though the option view >> SEH chain. As shown in figure 38.3

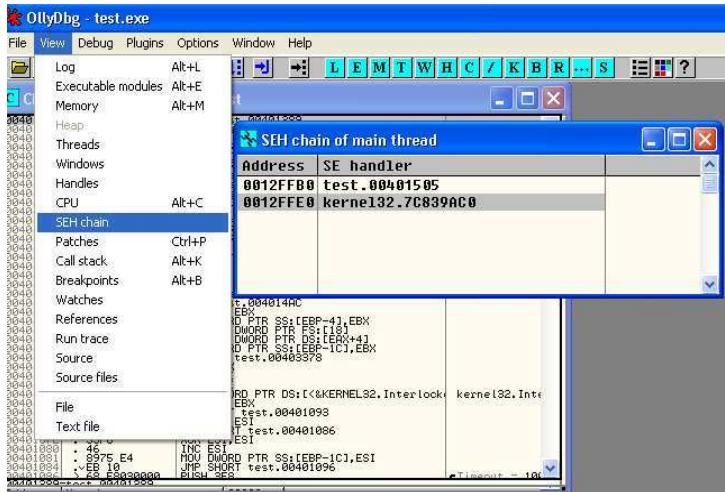


Figure 38.3 showing SEH chain

#### 4.6.1 Exploiting the SEH

As shown in figure 38.3, each exception register node is created on the stack and hence SEH vulnerability can be exploited. As already discussed each structure record contains the address of the handler function and pointer to the next record. The handler function address is referred and it is called when any exception occurs while executing any instruction. By overwriting the exception handler routine function address that is stored in the stack and flow of the application can be changed and further it can result in execution of malicious code. The code in figure 38.4 is of sample vulnerable application with SEH implementation

```
#include <windows.h>
void main(int argc, char* argv[])
{
    int i = 6.j;
    char buffer[10] = {0};
    __try
    {
        strcpy(buffer, argv[1]);
        i = i + 3;
        j = i % 9;
        i = i / j;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Exception Zero");
    }
}
```

Figure 38.4 showing sample SEH application

The figure 38.5 shows the status of the stack after creating and registering the exception handler node in SEH chain. As shown in the figure 38.5 the node is created in the stack. Here the vulnerability is application is coping the command line parameter passed to an application in a stack buffer without verifying the length of the buffer which can lead to buffer overflow. The figure 38.5 also shows where the passed command line parameter is getting stored in the buffer.

Figure 38.6 shows how this vulnerability is exploited in such a way that it overwrites the function address of exception handler routine that can further lead in changing the execution flow of an application. Here by providing very long string of “AAAA.....” overwite the stack location that stores the value of the exception hander functions address with the value 0x41414141. Now if this application raises any exception then rather than executing the original exception handler location 0x41414141 will get invoked. Instead of writing “AAA...” exploit code can be written which can lead to the execution of the code.

```

00401010 var_10= dword ptr -10h
00401010 var_4= dword ptr -4
00401010 arg_4= dword ptr 0Ch
00401010
00401010 push esp
00401011 mov ebp, esp
00401013 push 0FFFFF0h
00401015 push offset unk_420030
00401016 push offset unknown_librname_1 ; Micros
0040101F mov eax, large fs:0
00401025 push eax
00401026 mov large fs:0, esp
0040102D add esp, 0FFFFFFFA4h
00401030 push ebx
00401031 push esi
00401032 push edi
00401033 mov [ebp+var_18], esp
00401036 lea edi, [ebp+var_6C]
00401039 mov ecx, 15h
0040103E mov eax, 0CCCCCCCCh
00401043 rep stosd
00401045 mov [ebp+var_1C], 6
0040104C mov [ebp+Dest], 0
00401050 xor eax, eax
00401052 mov [ebp+var_28], eax
00401055 mov [ebp+var_27], eax
00401058 mov [ebp+var_23], al
0040105B mov [ebp+var_4], 0
00401062 mov ecx, [ebp+arg_4]
00401065 mov edx, [ecx+4]
00401068 push edx
00401069 lea eax, [ebp+Dest]
0040106C push eax
0040106D call _strupr
00401072 add esp, 0
00401075 mov ecx, [ebp+var_1C]

```

IDA View-ESP

\* 0012FF08 unk\_12FF08 dd 1B1CDCh ; ...  
\* 0012FF0C dd 00000000h  
\* 0012FF10 dd 7FFDE000h  
\* 0012FF14 ; [BEGIN OF STACK FRAME \_main\_0. PRESS KE  
\* 0012FF18 var\_6C db 00Ch  
\* 0012FF15 db 00Ch , 00Ch , 00Ch ; :|||  
\* 0012FF16 db 00CCCCCCCCh ; :|||  
\* 0012FF17 dd 00CCCCCCCCh ; :|||  
\* 0012FF18 dd 00CCCCCCCCh ; :||| Buffer location where strcpy  
\* 0012FF19 dd 00CCCCCCCCh ; :||| copies the buffer  
\* 0012FF1A dd 00CCCCCCCCh ; :|||  
\* 0012FF1B dd 00CCCCCCCCh ; :|||  
\* 0012FF1C dd 00CCCCCCCCh ; :|||  
\* 0012FF1D dd 00CCCCCCCCh ; :|||  
\* 0012FF1E dd 00CCCCCCCCh ; :|||  
\* 0012FF1F dd 00CCCCCCCCh ; :|||  
\* 0012FF20 dd 00CCCCCCCCh ; :|||  
\* 0012FF21 dd 00CCCCCCCCh ; :|||  
\* 0012FF22 dd 00CCCCCCCCh ; :|||  
\* 0012FF23 dd 00CCCCCCCCh ; :|||  
\* 0012FF24 dd 00CCCCCCCCh ; :|||  
\* 0012FF25 dd 00CCCCCCCCh ; :|||  
\* 0012FF26 dd 00CCCCCCCCh ; :|||  
\* 0012FF27 dd 00CCCCCCCCh ; :|||  
\* 0012FF28 dd 00CCCCCCCCh ; :|||  
\* 0012FF29 dd 00CCCCCCCCh ; :|||  
\* 0012FF2A dd 00CCCCCCCCh ; :|||  
\* 0012FF2B dd 00CCCCCCCCh ; :|||  
\* 0012FF2C dd 00CCCCCCCCh ; :|||  
\* 0012FF2D dd 00CCCCCCCCh ; :|||  
\* 0012FF2E dd 00CCCCCCCCh ; :|||  
\* 0012FF2F dd 00CCCCCCCCh ; :|||  
\* 0012FF30 dd 00CCCCCCCCh ; :|||  
\* 0012FF31 dd 00CCCCCCCCh ; :|||  
\* 0012FF32 dd 00CCCCCCCCh ; :|||  
\* 0012FF33 dd 00CCCCCCCCh ; :|||  
\* 0012FF34 dd 00CCCCCCCCh ; :|||  
\* 0012FF35 dd 00CCCCCCCCh ; :|||  
\* 0012FF36 dd 00CCCCCCCCh ; :|||  
\* 0012FF37 dd 00CCCCCCCCh ; :|||  
\* 0012FF38 dd 00CCCCCCCCh ; :|||  
\* 0012FF39 dd 00CCCCCCCCh ; :|||  
\* 0012FF3A dd 00CCCCCCCCh ; :|||  
\* 0012FF3B dd 00CCCCCCCCh ; :|||  
\* 0012FF3C dd 00CCCCCCCCh ; :|||  
\* 0012FF3D dd 00CCCCCCCCh ; :|||  
\* 0012FF3E dd 00CCCCCCCCh ; :|||  
\* 0012FF3F dd 00CCCCCCCCh ; :|||  
\* 0012FF40 dd 00CCCCCCCCh ; :|||  
\* 0012FF41 dd 00CCCCCCCCh ; :|||  
\* 0012FF42 dd 00CCCCCCCCh ; :|||  
\* 0012FF43 dd 00CCCCCCCCh ; :|||  
\* 0012FF44 dd 00CCCCCCCCh ; :|||  
\* 0012FF45 dd 00CCCCCCCCh ; :|||  
\* 0012FF46 dd 00CCCCCCCCh ; :|||  
\* 0012FF47 dd 00CCCCCCCCh ; :|||  
\* 0012FF48 dd 00CCCCCCCCh ; :|||  
\* 0012FF49 dd 00CCCCCCCCh ; :|||  
\* 0012FF4A dd 00CCCCCCCCh ; :|||  
\* 0012FF4B dd 00CCCCCCCCh ; :|||  
\* 0012FF4C dd 00CCCCCCCCh ; :|||  
\* 0012FF4D dd 00CCCCCCCCh ; :|||  
\* 0012FF4E dd 00CCCCCCCCh ; :|||  
\* 0012FF4F dd 00CCCCCCCCh ; :|||  
\* 0012FF50 dd 00CCCCCCCCh ; :|||  
\* 0012FF51 dd 00CCCCCCCCh ; :|||  
\* 0012FF52 dd 00CCCCCCCCh ; :|||  
\* 0012FF53 dd 00CCCCCCCCh ; :|||  
\* 0012FF54 aitest db 'Test',0  
\* 0012FF55 var\_27 dd 0  
\* 0012FF56 var\_23 dd 0  
\* 0012FF57 dw 0CCCCh ; :|||  
\* 0012FF58 var\_20 dd 0CCCCCCCCh  
\* 0012FF59 var\_1C dd 6  
\* 0012FF60 var\_18 dd offset unk\_12FF08  
\* 0012FF61 saved\_fp dd 83h ; :|||  
\* 0012FF62 saved\_fp dd 481299h ; unknown\_librname\_1  
\* 0012FF63 saved\_fp dd 420B30h ; rdata:unk\_420B30  
\* 0012FF64 var\_4 dd 0  
\* 0012FF65 saved\_fp dd 12FFC0h ; Stack[00001220]:  
\* 0012FF66 retaddr dd 481459h ; start+E9  
\* 0012FF67 saved\_fp dd 0  
\* 0012FF68 saved\_fp dd 2

Figure 38.5 Showing the status of stack

```

00401010 var_10= dword ptr -10h
00401010 var_4= dword ptr -4
00401010 arg_4= dword ptr 0Ch
00401010
00401010 push esp
00401011 mov ebp, esp
00401013 push 0FFFFF0h
00401015 push offset unk_420030
00401016 push offset unknown_librname_1 ; Micros
0040101F mov eax, large fs:0
00401025 push eax
00401026 mov large fs:0, esp
0040102D add esp, 0FFFFFFFA4h
00401030 push ebx
00401031 push esi
00401032 push edi
00401033 mov [ebp+var_18], esp
00401036 lea edi, [ebp+var_6C]
00401039 mov ecx, 15h
0040103E mov eax, 0CCCCCCCCh
00401043 rep stosd
00401045 mov [ebp+var_1C], 6
0040104C mov [ebp+Dest], 0
00401050 xor eax, eax
00401052 mov [ebp+var_28], eax
00401055 mov [ebp+var_27], eax
00401058 mov [ebp+var_23], al
0040105B mov [ebp+var_4], 0
00401062 mov ecx, [ebp+arg_4]
00401065 mov edx, [ecx+4]
00401068 push edx
00401069 lea eax, [ebp+Dest]
0040106C push eax
0040106D call _strupr
00401072 add esp, 0
00401075 mov ecx, [ebp+var_1C]
00401078 add esp, 3
0040107B mov [ebp+var_1C], ecx
0040107E mov eax, [ebp+var_1C]
00401081 ret

```

IDA View-ESP

\* 0012FF08 unk\_12FF08 dd 1B1CDCh ; ...  
\* 0012FF0C dd 00000000h  
\* 0012FF10 dd 7FFDE000h  
\* 0012FF14 ; [BEGIN OF STACK FRAME \_main\_0. PRESS KE  
\* 0012FF18 var\_6C db 00Ch  
\* 0012FF15 db 00Ch , 00Ch , 00Ch ; :|||  
\* 0012FF16 db 00CCCCCCCCh ; :|||  
\* 0012FF17 dd 00CCCCCCCCh ; :|||  
\* 0012FF18 dd 00CCCCCCCCh ; :||| Overwritten the exception  
\* 0012FF19 dd 00CCCCCCCCh ; :||| handler address  
\* 0012FF1A dd 00CCCCCCCCh ; :|||  
\* 0012FF1B dd 00CCCCCCCCh ; :|||  
\* 0012FF1C dd 00CCCCCCCCh ; :|||  
\* 0012FF1D dd 00CCCCCCCCh ; :|||  
\* 0012FF1E dd 00CCCCCCCCh ; :|||  
\* 0012FF1F dd 00CCCCCCCCh ; :|||  
\* 0012FF20 dd 00CCCCCCCCh ; :|||  
\* 0012FF21 dd 00CCCCCCCCh ; :|||  
\* 0012FF22 dd 00CCCCCCCCh ; :|||  
\* 0012FF23 dd 00CCCCCCCCh ; :|||  
\* 0012FF24 dd 00CCCCCCCCh ; :|||  
\* 0012FF25 dd 00CCCCCCCCh ; :|||  
\* 0012FF26 dd 00CCCCCCCCh ; :|||  
\* 0012FF27 dd 00CCCCCCCCh ; :|||  
\* 0012FF28 dd 00CCCCCCCCh ; :|||  
\* 0012FF29 dd 00CCCCCCCCh ; :|||  
\* 0012FF2A dd 00CCCCCCCCh ; :|||  
\* 0012FF2B dd 00CCCCCCCCh ; :|||  
\* 0012FF2C dd 00CCCCCCCCh ; :|||  
\* 0012FF2D dd 00CCCCCCCCh ; :|||  
\* 0012FF2E dd 00CCCCCCCCh ; :|||  
\* 0012FF2F dd 00CCCCCCCCh ; :|||  
\* 0012FF30 dd 00CCCCCCCCh ; :|||  
\* 0012FF31 dd 00CCCCCCCCh ; :|||  
\* 0012FF32 dd 00CCCCCCCCh ; :|||  
\* 0012FF33 dd 00CCCCCCCCh ; :|||  
\* 0012FF34 dd 00CCCCCCCCh ; :|||  
\* 0012FF35 dd 00CCCCCCCCh ; :|||  
\* 0012FF36 dd 00CCCCCCCCh ; :|||  
\* 0012FF37 dd 00CCCCCCCCh ; :|||  
\* 0012FF38 dd 00CCCCCCCCh ; :|||  
\* 0012FF39 dd 00CCCCCCCCh ; :|||  
\* 0012FF3A dd 00CCCCCCCCh ; :|||  
\* 0012FF3B dd 00CCCCCCCCh ; :|||  
\* 0012FF3C dd 00CCCCCCCCh ; :|||  
\* 0012FF3D dd 00CCCCCCCCh ; :|||  
\* 0012FF3E dd 00CCCCCCCCh ; :|||  
\* 0012FF3F dd 00CCCCCCCCh ; :|||  
\* 0012FF40 dd 00CCCCCCCCh ; :|||  
\* 0012FF41 dd 00CCCCCCCCh ; :|||  
\* 0012FF42 dd 00CCCCCCCCh ; :|||  
\* 0012FF43 dd 00CCCCCCCCh ; :|||  
\* 0012FF44 dd 00CCCCCCCCh ; :|||  
\* 0012FF45 dd 00CCCCCCCCh ; :|||  
\* 0012FF46 dd 00CCCCCCCCh ; :|||  
\* 0012FF47 dd 00CCCCCCCCh ; :|||  
\* 0012FF48 dd 00CCCCCCCCh ; :|||  
\* 0012FF49 dd 00CCCCCCCCh ; :|||  
\* 0012FF4A dd 00CCCCCCCCh ; :|||  
\* 0012FF4B dd 00CCCCCCCCh ; :|||  
\* 0012FF4C dd 00CCCCCCCCh ; :|||  
\* 0012FF4D dd 00CCCCCCCCh ; :|||  
\* 0012FF4E dd 00CCCCCCCCh ; :|||  
\* 0012FF4F dd 00CCCCCCCCh ; :|||  
\* 0012FF50 dd 00CCCCCCCCh ; :|||  
\* 0012FF51 dd 00CCCCCCCCh ; :|||  
\* 0012FF52 dd 00CCCCCCCCh ; :|||  
\* 0012FF53 dd 00CCCCCCCCh ; :|||  
\* 0012FF54 aitest db 'Test',0  
\* 0012FF55 var\_27 dd 0  
\* 0012FF56 var\_23 dd 0  
\* 0012FF57 dw 4141411h ; :|||  
\* 0012FF58 var\_20 dd 4141411h ; :|||  
\* 0012FF59 var\_18 dd 4141411h ; :|||  
\* 0012FF5A var\_1C dd 4141411h ; :|||  
\* 0012FF5B var\_10 dd 4141411h ; :|||  
\* 0012FF5C saved\_fp dd 4141411h ; :|||  
\* 0012FF5D retaddr dd 4141411h ; :|||  
\* 0012FF5E UNKNOWN dd 420B30h ; rdata:unk\_420B30  
\* 0012FF5F var\_4 dd 0  
\* 0012FF60 saved\_fp dd 12FFC0h ; Stack[00001220]:  
\* 0012FF61 retaddr dd 481459h ; start+E9  
\* 0012FF62 saved\_fp dd 0  
\* 0012FF63 saved\_fp dd 2

Figure 38.6 Showing the exploitation of the vulnerability

## 4.7 Writing Exploits General Concepts

Writing exploits involve understanding vulnerabilities in an application. This section discusses the stack overflow and heap overflow which is important in writing exploits.

### 4.7.1 Stack Overflow Exploits

Figure 37.0 shows the stack organization of IA-32 Intel 32 Bit x86 architecture. The stack on IA-32 grows downwards. This is unlike SPARC architecture where the stack grows upwards. LIFO (Last in First Out) is used to push variables in the stack. The stack stores parameters, buffers and return address for the function.

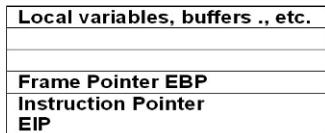


Figure 37.0 Stack organization of IA-32.

Figure 38.0 shows two buffers, buffer1 [] and buffer2 [] pushed on the stack. As can be seen, buffer1 [] was the first buffer pushed on to the stack and buffer2 [] was the second buffer pushed on to the stack.

If buffer1[] as shown in figure 38.0 has more elements than its size, it will end up writing in the buffer buffer2[]. EIP stores the return address. Overflowing buffer1[] will rewrite EIP as well and the attacker can control the memory address that is returned to the calling function.

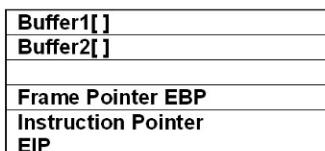


Figure 38.0 showing the buffer stored in the Stack for IA-32.

Once the control to process is reached, the next step involves diverting the control. This is accomplished by pointing the EIP to the payload. For successful exploitation, the payload has to be first injected in the buffer. Then the controlled EIP is directed to the payload to be executed.

#### 4.7.2 Injection Techniques

To execute an exploit, it is mandatory that the large buffer is inserted in the overflowable buffer which can be accomplished by automating buffer filling over the network or crafting a malicious file which is accessed by the vulnerable process.

#### 4.7.3 Optimizing the Injection Vector

The custom operational code (*opcode*) which is required to control the instruction pointer on the remote machine is called as injection vector. The objective of the injection vector is to ready the payload to be executed.

### 4.8 The Location of the Payload

The payload and the injection vector can be located at different places. However a stack can be used for both. When a stack is used for both the payload and the injection vector then if the size of the payload starts before the injection vector, it has to be ensured that no collision occurs between them. In case of collision, a jump has to be included in the payload such that the payload can jump over the injection code and can continue on the other side of the injection vector. The other option would be to place the payload at a location which is different from the injection vector. A possible candidate for storing the payload is the buffer where the program stores its information. Files on disk, environment variables controlled by a local user, environment variables which are passed within a web request and the user controlled fields within a network protocol are some of the places which can be used to store the payload. After injecting the payload, the instruction pointer will have to be loaded with the address of the payload. Storing the payload somewhere other than the stack provides the inherent advantage that the payload can be of a large size.

Once the payload has been injected, the task is to get the instruction pointer to load the address of the payload. The beauty of storing the payload somewhere other than the stack is that amazingly tight and difficult-to-exploit

buffer overflows suddenly become possible (e.g., we are free from constraints on the size of the payload). A single “off-by-one” error can still be used to take control of a computer.

After loading the payload, the next step involves executing the payload. The saved EIP, on the stack has to be modified so that it points to the modified payload. The next section discusses the techniques for jumping to the payload.

#### 4.8.1 Direct Jump (Guessing Offsets)

By using this method, the overflow code is jumped directly to the memory location. The *direct jump* means that the overflow code was told to jump directly to a specific location in memory. In this case, it might happen that the address of the stack may contain null characters so the payload has to be placed before the injector. This limits the available space for the payload. The address of the payload might not always be the same. Hence, a reasonable guess has to be made about the address. Generally, this method is preferred in UNIX as in UNIX the null character does not contain the address of the stack in UNIX. The direct jump is generally preferred when the payload is placed some where other than on the stack.

#### 4.8.2 Blind Return

The *ret* instruction causes the EIP to be loaded with the value in ESP, which points to the current stack location. When the *ret* is executed, the topmost value in the stack is loaded into the EIP resulting in the EIP pointing to the new code address. If the EIP is injected with a value which points to the *ret* instruction, then the value stored at the ESP is loaded in the ESI.

#### 4.8.3 pop Return

It might happen that the top of the stack does not point to the address of the buffer storing the exploit code. In such a scenario, the injected EIP can be set to point to a series of pop instructions which is followed by *ret*. Before a value can be used for the EIP register, a series of pop operations will result in popping the stack a number of times. Adding a series of pop techniques will work, if the address which is near the top of the stack points to within the attacker’s buffer. Before the useful address is reached, the attacker points to a series of pop instructions.

```
- pop EBX 5B  
- pop ECX 59  
- pop EAX 58  
- pop ESI 5E  
- pop EDI 5F  
- pop EBP 5D  
- ret      C3
```

#### 4.8.4 No Operation Sled

Injection of the direct address of payload requires guessing the exact location of payload in memory. It is quite possible that the payload is not always at the same address. It commonly occurs that the software package is recompiled on different systems with different compilers having different optimization techniques. To overcome the limitations of precise addresses of exploit code, NOP sleds are used. An NOP instruction does nothing, however it takes one byte of address space. These NOPs are added before the exploit code. Since the buffer now comprises NOPs followed by the exploit code, any address containing NOPs can be used. If the return address contains an address of any of the NOP instructions then firstly the NOP instructions are executed followed by the exploit code. It has to be noted that the larger the size of NOP sleds, the less precision is required in guessing the address of the payload.

#### 4.8.5 Call Register

This method is used when the register is loaded with the address which points to the exploit code. The EIP will have to be loaded with the instruction which enables the call to register. This method is commonly used in Windows based exploits as there are many commands at fixed addresses in the kernel32.dll. This method can be used at any process.

```
- call EAX FF D0  
- call EBX FF D3  
- call ECX FF D1  
- call EDX FF D2  
- call ESI FF D6  
- call EDI FF D7  
- call ESP FF D4
```

#### 4.8.6 Push Return

The push return method is used when the register is loaded with the address of the exploit code and the *call* instruction cannot be located. The other option will be to locate the push <register> which is followed by ret.

#### 4.8.7 Calculating Offset

If the attacker has access to a computer, exploit code can be compiled directly onto the computer. The injection code can calculate its base and assumed that the program which is being attacked has the same base. To execute exploit code, the attacker in such a scenario will have to specify the offset from this address for a direct jump, i.e., the *base+offset* value of the attacking code is similar to the victim code.

### 4.9 Conclusion

Poorly written code can lead to vulnerable software. The most commonly found vulnerabilities are buffer overflow, heap overflow, integer overflow and format string vulnerability. In most cases, the source code of the software may not be available so binary analysis is required. Format string vulnerability in software can be identified by monitoring the printf family of functions in IDA pro. The number of arguments passed to the function should be equivalent to the format specifiers. For stack overflow, the stack size has to be monitored first. This can be identified by monitoring SUB ESP instructions in the assembly code of stack. The SUB ESP instruction will give the size of stack. After the stack size is determined to check for buffer overflow, generally the LEA instruction can be monitored. The operand to the LEA instruction will identify the usage of the allocated space. Off-by-one overflow happens when the string is not terminated by a null character. It might lead to other important data like EBP, which might be used later. Integer overflow happens because the values held in a numeric data type are limited by the data type's size in bytes. For identifying integer overflows, it has to be first identified in the assembly if the buffer length is signed or unsigned. If the instruction JG /JLE/JGE is used, then it denotes that the buffer length/length is treated as a signed integer by the compiler, if the instruction JA/JBE/JAE/JNB is used then the compiler is treating buffer length/length as the unsigned integer. Heaps are stored in memory as a linked list with each pointer having a pointer to the next one. Heap overflow results in rewriting the pointer to the next block. Heap overflow may not result in a crash.

# Fundamental of Reverse Engineering

## 5.0 Introduction

Reverse engineering is a technique to find the design of a software from its binary. Vulnerability researchers, academics, and security professionals use this technique to learn design of software for various proposes such as writing an anti privacy wrapper softwares. Reverse engineering is also used in malware analysis, copyright and patent litigation, discovery of undocumented APIs, malware creation, recovery of data from the proprietary file formats. If the aim of reverse engineering is to copy or duplicate programs, it may result in a copyright violation. However in most of the cases, the licensed use of software prohibits reverse engineering.

This chapter discusses the anti-reversing techniques, which include concept of disassembly, anti debugging, and virtual machine detection. This is followed by a discussion on the packers and their protection mechanism. Packers, which disassemble the binaries, also prevent reverse engineering of software. The chapter concludes with the unpacking mechanisms. Some of these sections also appeared in the book titled “Vulnerability Analysis and Defense for the Internet”.

### 5.1 Anti-Reversing Method

Anti-reversing methods are applied in malicious software, licenses copy protections and digital rights management. Anti-reversing methods are included in binary to increase the complexity of reversing the binary file. Anti-reversing methods evade the tools that are used by the reverse engineers by exploiting its design, implementation, functionalities or vulnerabilities. Although anti-reversing methods reduce the efficiency of an application, increasing its code size and sometimes affects the robustness of an application; however anti reversing is required to prevent from binary analysis. Anti-reversing methods are commonly used for performing anti-disassembly, self-modifying codes, anti debugging, and for virtual machine detection.

### 5.2.1 Anti Disassembly

Anti Disassembly is an anti-reversing methods used to evade disassemblers. This enables the disassemblers to generate incorrect disassembled code. The knowledge of anti disassembly will help to better understand the working of disassembler. There are basically two types of disassemblers. They are Linear Sweep Disassembler and Recursive Traversal Disassembler.

#### 5.2.1.1 Linear Sweep Disassembler

Linear Sweep Disassembler as shown in figure 1 starts it's processing from the beginning of the software binary and decodes the instructions in sequence until it reaches the end of the binary.

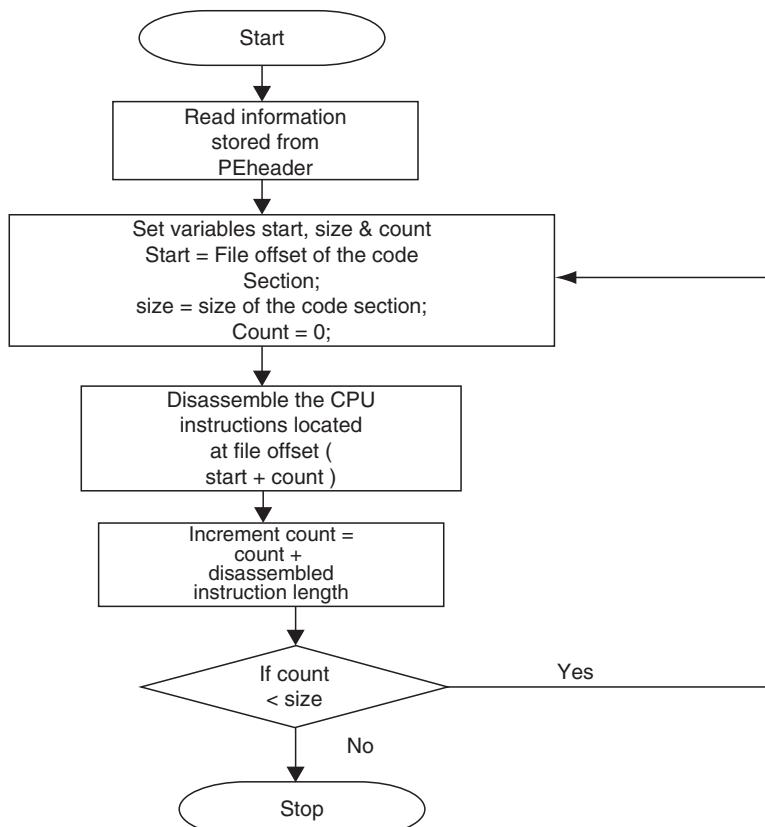


Figure 1: Working Logic of Liner Sweep Disassembler

Although the method is quite fast, it cannot be used to detect and handle data/code mix-up regions in the executable. Data code mix-up region is a location in the executable binary, where certain data bytes are present within the instruction byte. One of the very common examples of data code as shown in figure 2 mix-up is a switch table generated by the compiler, while using switch statements

00401006	. C745 FC 000000	MOU DWORD PTR SS:[EBP-4],0
0040100D	. 8B45 08	MOU EAX,DWORD PTR SS:[EBP+8]
00401010	. 8945 F8	MOU DWORD PTR SS:[EBP-8],EAX
00401013	. 8B4D F8	MOU ECX,DWORD PTR SS:[EBP-8]
00401016	. 83E9 01	SUB ECX,1
00401019	. 894D F8	MOU DWORD PTR SS:[EBP-8],ECX
0040101C	. 837D F8 03	CMP DWORD PTR SS:[EBP-8],3
00401020	.~77 2E	JAE SHORT license_.00401050
00401022	. 8B55 F8	MOU EDX,DWORD PTR SS:[EBP-8]
00401025	. FF2495 541040	JMP DWORD PTR DS:[EDX*4+401054]
0040102C	> 8B45 08	MOU EAX,DWORD PTR SS:[EBP+8]
0040102F	. 83C0 01	ADD EAX,1
00401032	. 8945 FC	MOU DWORD PTR SS:[EBP-4],EAX
00401035	> 8B4D 08	MOU ECX,DWORD PTR SS:[EBP+8]
00401038	. 83C1 01	ADD ECX,1
0040103B	. 894D FC	MOU DWORD PTR SS:[EBP-4],ECX
0040103E	> 8B55 08	MOU EDX,DWORD PTR SS:[EBP+8]
00401041	. 83C2 01	ADD EDX,1
00401044	. 8955 FC	MOU DWORD PTR SS:[EBP-4],EDX
00401047	> 8845 08	MOU EAX,DWORD PTR SS:[EBP+8]
0040104A	. 83C0 01	ADD EAX,1
0040104D	. 8945 FC	MOU DWORD PTR SS:[EBP-4],EAX
00401050	> 8BE5	MOU ESP,EBP
00401052	. 5D	POP EBP
00401053	. C3	RETN
00401054	. 2C104000	DD license_.0040102C
00401058	. 35104000	DD license_.00401035
0040105C	. 3E104000	DD license_.0040103E
00401060	. 47104000	DD license_.00401047
00401064	[\$ 55	PUSH EBP
00401065	. 8BEC	MOU EBP,ESP

Figure 2: Data Code Mix-up

As shown in figure 2, the address values from 00401054 to 00401064 are non-executable data bytes, which are present before and after the executable instruction bytes. In order to optimize the performance of the application, some compilers add data within the code block, making it difficult for Linear Sweep Disassembler to disassemble the code. The Linear Sweep Disassembler assumes every byte as an executable instruction byte, failing to determine whether the byte has to be treated as data or instruction. This results in generation of incorrect assembly instructions. A Linear Sweep disassembler

will generate assembly code (as shown in figure 3) in the place of switch table, as the Linear Sweep disassembler will treat jump table as an instruction byte and not as a data byte.

```
sub    al, 10h
inc    eax
add    ds:3E004010h, dh
adc    [eax+0], al
inc    edi
adc    [eax+0], al
```

Figure 3: Disassembled Jump Table

#### 5.2.1.2 Recursive Traversal Disassembler

Recursive Traversal Disassemblers are smart disassemblers compared to the Linear Sweep disassemblers. Similar to Linear Sweep Disassemblers, Recursive Traversal Disassemblers start processing the binary from the beginning of a file. In contrast to the Linear Sweep Disassembler, the order of processing of bytes in the software binary file is not sequential. The disassembling flow depends on the control flow of the program. The next byte to be processed depends on the last processed instruction. If the last processed instruction is jump/call then jump/call instruction will be processed else the next consecutive instruction will be processed. The flow chart shown in Figure 5 shows the working of Recursive Traversal Disassembler

Due to its complex logic, recursive traversal disassemblers are not as fast as Linear Sweep disassemblers. The disassembled output generated by recursive traversal disassemblers is more accurate compared to the results generated by a Linear Sweep disassembler. The control flow processing strategy helps recursive disassembler in easily identifying data code mix-up.

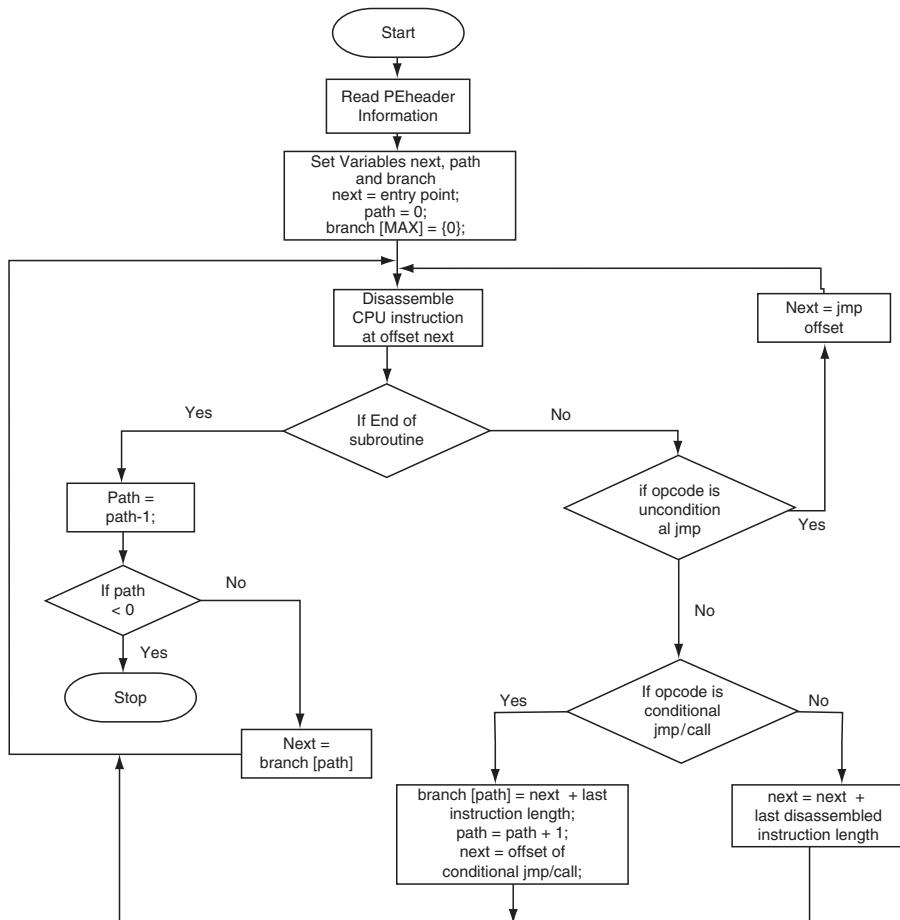


Figure 5 Working Logic of Recursive Traversal Disassembler

### 5.2.1.3 Evasion of Disassembler

In order to evade any disassembler (Linear Sweep or recursive) data-code mix-up blocks has to be added in such a way that the code disassembler gets confused in identifying the correct data byte, at the time of disassembling. However, it should be ensured that it does not affect the execution or behavior of the program.

The Linear Sweep disassembler can be successfully evaded by using the code shown in figure 6.0. Linear Sweep disassemblers are easier to evade.

```

__asm {
    __emit 0xEB;
    __emit 0x01;
    __emit 0x0F;
}

```

Figure 6:0 Showing the Linear Sweep Disassembler.

The equivalent assembly code for the figure shown in 6.0 is shown in figure 7.0

```

xxxx1 Jmp xxxx4
 0x0F
xxxx4 .....

```

Figure 7.0 Showing the resulting assembly of the code in shown in figure 6.0

The code shown in figure 7.0 will insert short jumps of one byte within an application code. The figure 8.0 shows how the assembly block incorporates the short jump instruction.

.text:00401024	mov [ebp+var_5C4], 0
.text:0040102E	jmp short loc_401031
.text:0040102E	db 0Fh
.text:00401031	cmp [ebp+argc], 4
.text:00401035	jge short loc_401056

Figure 8.0 Assembly block incorporating the short jump instruction

If the code shown in figure 8.0 is opened in any Linear Sweep Disassembler, it will generate disassembled output as shown in figure 9.0.

.text:00401024	mov [ebp+var_5C4], 0
.text:0040102E	jmp short near ptr loc_401030+1
.text:00401030	jnb near ptr 7D4418B3h
.text:00401036	pop ds

Figure 9.0 Assembly block generated by Linear Sweep Disassembler

It can be inferred that the Linear Sweep Disassembler is not able to disassemble last two instructions correctly. As shown in figure 9.0 rather than interpreting byte at address **0040102E** as data byte and next two instructions as cmp and jge, it is interpreting the next two instructions as jnb and pop. So these kind of short jumps are well enough to evade Linear Sweep Disassembler. These short jumps are basically inserting a data byte in between the application code. The Linear Sweep Disassembler works on an assumption that every byte in code section belongs to executable instruction. Hence, it is interpreting the byte at address **0040102E** as a part of CPU instruction. However, as it is a non-executable data byte at the time of execution, the control never comes to the instruction **0040102E**, resulting in a

wrong disassembly. However, the disassembled output generated by any Recursive Traversal Disassembler (like IDA pxro or ollydbg) easily detects this short jump technique and generates the following correct sequence of instruction.

```
.text:00401024      mov    [ebp+var_5C4], 0
.text:0040102E      jmp    short loc_401031
.text:0040102E ; -----
.text:00401030      db 0Fh
.text:00401031 ; -----
.text:00401031
.text:00401031 loc_401031:           ; CODE XREF: _main+16j
.text:00401031      cmp    dword ptr [ebp+8], 4
.text:00401035      jge    short loc_401056
```

Figure 10.0 Assembly code generated by the Recursive Traversal Disassembler.

As the processing of Recursive Traversal Disassembler depends on the control flow of the application, it can easily identify the non-executable data bytes. Evading Recursive Traversal Disassembler can be slight challenging. It can be done using opaque predicate. Opaque predicates are false condition statements, which appear to be conditional but in reality it is unconditional. The conditional branch splits the flow into two paths. The opaque predicates, inserts condition in such a way that one path leads to the real code and the other path to the junk code. The junk code never gets executed.

```
asm{
    mov eax, 4
    cmp eax, 6
    je Junk
    jne real_code
    Junk:
        .emit 0xf
    real_code
}
```

Figure 11.0 Showing Opaque Predicate

As shown in figure 11.0 the first instruction in this block assigns value 4 to eax register. The next instruction compares the value stored in eax (which is 4) with the constant value 6. The comparison will never be equal. Hence, every time the comparison fails, the control will be passed to the code under the label real\_code. This is an ideal example of opaque predicate. Although the above code looks like a conditional jump, in reality, the code will only have a fixed flow.

```

text:0040102E    mov eax, 4
.text:00401030    cmp eax, 6
.text:00401032    je 00401036
.text:00401034    jne 00401037
.text:00401036    db 0x0F
.text:00401037    mov ebx,ecx
.text:00401039    cmp eax,ebx
.text:0040104B    NOP

```

Figure 12.0 Code with opaque predicate block.

Code in figure 12.0 shows the assembly code of a sample application containing opaque predicate block. This code when opened in any recursive traversal disassembler will give the output as shown in figure shown in figure 13.0

```

.text:0040102E    mov eax, 4
.text:00401030    cmp eax, 6
.text:00401032    je 00401036
.text:00401034    jne 00401037
.text:00401036    JPO 91034C15

```

Figure 13.0 Code showing the output by recursive traversal disassembler.

As shown in figure 13 the recursive traversal disassembler is not able to disassemble the last two instructions correctly. Rather than interpreting the byte at address **00401036** as data and next instructions as mov and cmp, recursive traversal disassemblers interprets the instructions as je, jne and jpo. Opening the code in linear sweep disassembler also gives incorrect disassembled output. Hence, opaque instruction can be used to evade both the Linear Sweep and the Recursive Traversal disassembler. The value of eax and flag register might change, which may lead to the execution of junk code or some other code. Execution of junk code may affect the control flow and the data flow of the program. Hence, to design an opaque instruction, it is required that the flow of instructions should be as desired and value of registers should ensure the correctness of control flow and the data flow.

```

__asm{
    jno    condition1
    jo     condition2
    condition1
    jo     Junk1
    jno    Real_code
Junk1
    db 0Fh
    condition2
    jno    Junk2
    jo     Real_code
Junk2
    db 0Fh
    Real_code
}

```

Figure 14.0 Flag based opaque predicate.

The figure 14.0 shows one of the opaque instructions in which the value of register is not getting altered. As shown in figure 14.0 the first two

instructions are checking overflow flags of the flag register and if it is set then the control will jump to condition 2 or it will jump to condition 1. The instruction present here is again checking for overflow. However, it is ensured that the control is passed to the `Real_code`.

```

> 6A 64    PUSH 64
. 68 F4C4000 PUSH org_lice.0040C4F4
. 6A 67    PUSH 67
. 8B45 08    MOV EAX,DWORD PTR SS:[EBP+8]
. 50    PUSH EAX
. FF15 D0804000 CALL DWORD PTR DS:[<&USER32.LoadStringA]
. 6A 64    PUSH 64
. 68 90C4000 PUSH org_lice.0040C490
. 6A 6D    PUSH 6D
. 8B4D 08    MOV ECX,DWORD PTR SS:[EBP+8]
. 51    PUSH ECX
. FF15 D0804000 CALL DWORD PTR DS:[<&USER32.LoadStringW]
. 8B55 08    MOV EDX,DWORD PTR SS:[EBP+8]

```

Annotations:

- `PUSH 64`: Count = 64 (100.)
- `PUSH org_lice.0040C4F4`: Buffer = org\_lice.0040C4F4
- `RsrcID = STRING "license_check"`
- `PUSH 67`: hInst
- `MOV EAX,DWORD PTR SS:[EBP+8]`: LoadStringA
- `PUSH 6D`: Count = 64 (100.)
- `PUSH org_lice.0040C490`: Buffer = org\_lice.0040C490
- `RsrcID = STRING "LICENSE_CHECK"`
- `MOV ECX,DWORD PTR SS:[EBP+8]`: hInst
- `PUSH ECX`: LoadStringW
- `CALL DWORD PTR DS:[<&USER32.LoadStringA]`: Count = 64 (100.)
- `CALL DWORD PTR DS:[<&USER32.LoadStringW]`: Buffer = org\_lice.0040C490
- `MOV EDX,DWORD PTR SS:[EBP+8]`: RsrcID = STRING "LICENSE\_CHECK"

Figure 15.0 shows the disassembled code with and without anti disassembly macro.

```

. 57    PUSH EDI
. 6A 64    PUSH 64
. 73 05    JNB SHORT 4license.00401012
. 72 08    JB SHORT 4license.00401017
. EB 09    JNP SHORT 4license.00401011
> 0F72    ???
? 0273 06    ADD DH,BYTE PTR DS:[EBX+6]
> 0F73    ???
? 0272 01    ADD DH,BYTE PTR DS:[EDX+1]
. 0F68F4    PUNPCKHWD MM6,MM4
. C400 00    LES EAX,FWORD PTR DS:[EAX]
. 73 05    JNB SHORT 4license.00401028
. 72 08    JB SHORT 4license.0040102D
. EB 09    JNP SHORT 4license.00401027
> 0F72    ???
? 0273 06    ADD DH,BYTE PTR DS:[EBX+6]
> 0F73    ???
? 0272 01    ADD DH,BYTE PTR DS:[EDX+1]
. 0F6A67 73    PUNPCKHDQ MM4,WORD PTR DS:[EDI+73]
. 05 7208EB00    ADD EAX,0EB0872
> 0F72    ???
? 0273 06    ADD DH,BYTE PTR DS:[EBX+6]
> 0F73    ???
? 0272 01    ADD DH,BYTE PTR DS:[EDX+1]
> 0F88 45087305    JPO 0508188F
. 72 08    JB SHORT 4license.00401054
. EB 00    JNP SHORT 4license.0040104E
> 0F72    ???
? 0273 06    ADD DH,BYTE PTR DS:[EBX+6]
> 0F73    ???

```

Annotations:

- `PUSH EDI`: Unknown command
- `PUSH 64`: Unknown command
- `JNB SHORT 4license.00401012`: Modification of segment register
- `JB SHORT 4license.00401017`: Unknown command
- `JNP SHORT 4license.00401011`: Unknown command
- `ADD DH,BYTE PTR DS:[EBX+6]`: Unknown command
- `ADD DH,BYTE PTR DS:[EDX+1]`: Unknown command
- `PUNPCKHWD MM6,MM4`: Modification of segment register
- `LES EAX,FWORD PTR DS:[EAX]`: Unknown command
- `JNB SHORT 4license.00401028`: Unknown command
- `JB SHORT 4license.0040102D`: Unknown command
- `JNP SHORT 4license.00401027`: Unknown command
- `ADD DH,BYTE PTR DS:[EBX+6]`: Unknown command
- `ADD DH,BYTE PTR DS:[EDX+1]`: Unknown command
- `PUNPCKHDQ MM4,WORD PTR DS:[EDI+73]`: Unknown command
- `ADD EAX,0EB0872`: Unknown command
- `ADD DH,BYTE PTR DS:[EBX+6]`: Unknown command
- `JPO 0508188F`: Unknown command
- `JB SHORT 4license.00401054`: Unknown command
- `JNP SHORT 4license.0040104E`: Unknown command
- `ADD DH,BYTE PTR DS:[EBX+6]`: Unknown command

Figure 16.0 showing the code using anti disassembly macro.

The figure 15.0 shows the sample code of an application that does not uses any anti-disassembly method. Figure 16.0 shows the equivalent code using anti-disassembly macro to evade the disassembler like OllyDbg.

### 5.2.2 Self-modifying code

Self-modifying code (SMC) is one of the methods to prevent application from reverse engineering. This method can further be extended to some advanced

method like polymorphism and metamorphism to prevent disassembly from generating the original code of an application. SMC is the method in which the application itself modifies its instruction at the time of execution. By incorporating SMC application developers hide their protected code from disassembler.

The use of SMC in protecting the application from getting reverse engineered is explained with the help of the following case study. The figure 17.0 shows a sample function called Check Number, which checks whether the integer is odd or even.

```
void checknumber(unsigned int number)
{
    _asm push eax;
    _asm mov eax,[ebp+8];
    _asm and eax,1;
    _asm cmp eax,1;
    _asm je odd;
    printf("Even number");
    _asm pop eax;
    return;
odd:
    printf("odd number");
    _asm pop eax;
    return;
}
```

Figure 17.0 showing a sample function which checks if integer is odd or even.

The figure 18.0 shows the sample code, which can be used to prevent the disassembler from the disassembling the highlighted code shown in 17.0

The code shown in the figure 18.0 when opened in any disassembler will not show the instructions, which are highlighted in figure 17.0. The application does not store the instruction set but it is XORing the bytes with the key 0xBA��. At the time of execution the code shown in 19.0 block decodes the instructions and revives it back to its original state so that it can be executed correctly. Hence, it becomes tough for a disassembler to disassemble the code.

```

void CheckNumber(unsigned int i)
{
    unsigned int size = SIZE_OF_BLOCK_IN_BYTES_NEED_TO_BE_PROTECTED_;
    unsigned int _ = STARTING_VIRTUAL_ADDRESS_OF_BLOCK;
    _asm push ecx;
    _asm push ebx;
    _asm push eax;
    _asm mov eax,[ebp-8];
    _asm mov ecx,[ebp-4];
Loop1:
    _asm mov ebx,[eax];
    _asm xor ebx,0xBADB;
    _asm mov [eax],ebx;
    _asm sub ecx,4;
    _asm add eax,4;
    _asm cmp ecx,0;
    _asm je Start1;
    _asm jne Loop1;

Start1:// STARTING_VIRTUAL_ADDRESS_OF_BLOCK
    _asm _emit 0x50;
    _asm _emit 0xFF;
    _asm _emit 0x08;
    _asm _emit 0x83;
    _asm _emit 0x3B;
    _asm _emit 0xBB;
    _asm _emit 0x83;
    _asm _emit 0xF8;
    _asm _emit 0xDA;
    _asm _emit 0x2A;
    _asm _emit 0x90;
    _asm _emit 0x90;

End1:
    _asm je odd;
    printf("Even number");
    _asm pop eax;
    _asm pop ebx;
    _asm pop ecx;
    return;
odd:
    printf("odd number");
    _asm pop eax;
    _asm pop ebx;
    _asm pop ecx;
}

```

Figure 18.0 Sample function with SCM implementation.

```

    _asm mov ebx,[eax];
    _asm xor ebx,0xBADB;
    _asm mov [eax],ebx;
    _asm sub ecx,4;
    _asm add eax,4;
    _asm cmp ecx,0;
    _asm je Start1;
    _asm jne Loop1;

```

Figure 19.0 Decoding routines.

The processing unit block (decryption block length) of the decryption routine is DWORD (four bytes). So three more nops are added to make the size a multiple of four.

Dynamic/Runtime decryption and encryption is another method, which works similar to that of SMC. This is used for anti-reversing. Application developer needs to add certain predefined macros in their application code, which they want to protect from getting reversed.



Figure 20.0 showing template of Dynamic Encryption\Decryption

As shown in figure 20.0 CRYPT\_BEGIN and CRYPT\_END are the macros used to select a block of code, which has to be protected from getting reversed. At the time of packing or protecting, binary packer will encrypt the selected block of code and insert certain bytes, to transfer control to the routine that is responsible for decrypting and encrypting the byte instruction before and after its execution. As the instruction bytes are encrypted, the executable in disassembler will result in some junk instructions at the time of opening. However, to analyze the code protected by dynamic encryption and decryption, a hardware break point needs to be applied to the protected instruction without breaking the encryption algorithm.

As the decryption routine is called first, it decrypts the protected instruction and passes the control to the decrypted instruction byte, thereby applying the hardware break point to the protected code, resulting in termination of the execution. This retrieves the original code.

Polymorphism is another technique, which can be used by the virus writers to evade virus-detecting software. Polymorphism is the technique in which the virus code in each infected machine is represented differently/in a unique way. Signature based antivirus, searches for unique pattern. Changing the pattern of bytes for each infection makes the job more complicated. The concept being for every new victim a new key is generated and the body of the virus is encrypted with the help of the new key. For each infection, this makes byte pattern completely different. More advanced polymorphic engines not only change the key for protecting the sensitive code - they also change the algorithm of encryption and decryption, so that they don't get detected on the basis of encryption or decryption routine. Apart from this a good polymorphic engine will have the ability to generate different set of instructions, which do same work. For a code, it is possible to have n number of equivalent codes, which will perform the same operation but will look different.

```
Mov ebx,10;  
Cmp ebx,ecx;
```

Figure 21.0 showing instructions

For example, the code shown in figure 21.0 is equivalent to the code shown in figure 22.0

```
cmp ecx,10;  
Mov ebx,10;
```

Figure 22.0 showing instructions equivalent to 20.0

A good polymorphic engine also generates calls to dummy routines, conditional jumps and junk instructions. For hiding the control flow and the data flow of the program, various polymorphic engine generators insert calls to dummy routine, which tries to evade reverse engineering. Anti-debugging instructions are basically used to detect the presence of debugger and if a debugger is found then the program tries to evade it. Some of the most common anti-debugging techniques are discussed in the anti-debugging section. A combination of these techniques can make reversing of the decryption routine a complex task.

### 5.2.3 Virtual Machine Obfuscation

One of the most effective and powerful ways to achieve obfuscation is by implementing virtual machine. Basically in this technique the protected instruction sets are translated into P-codes, which are interpreted and executed at the runtime environment of virtual machine. Complete virtual machine implementation is required to reverse the protected code. The virtual machine provides the inherent advantage of modifying the standard CPU instruction present in executable into some customized form, which can be interpreted and executed only at the run time. Since the protected instruction does not have standard CPU instruction in the executable,, the debugger will not recognize and disassemble these instructions. At the time of execution, these customized instructions are executed by customized runtime environment. Hence, it does not require decoding these protected instructions back to the original form.

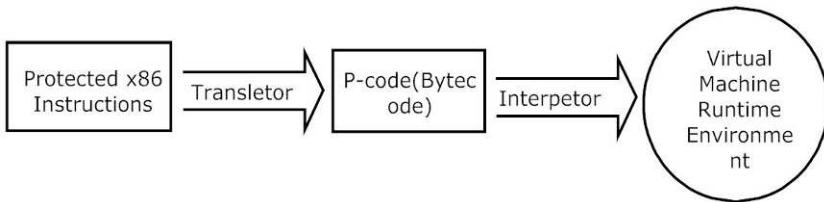


Figure 23.0 showing virtual machine implementation.

## 5.3 Anti Debugging Techniques

This section discusses about some of the debugger checks. Debugger Checks present in the executable helps to detect the availability of debugger in order to prevent the application from getting debugged. IsDebuggerPresent window API is one of the API that can be used to find whether the application is getting debugged or not. The return type of this function is Boolean, true in case the application is debugged else false. So the simplest method is to call this API and check for the return value. The code shown in the figure 23.0 shows the details of the API.

```

MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]
MOVZX EAX,BYTE PTR DS:[EAX+2]
RETN
    
```

Figure 24.0 code showing the details of Iddebugger API

The First statement MOV EAX, DWORD PTR FS:[18] is moving the value of current executing thread environment block into EAX register. The instruction EAX, DWORD PTR DS:[EAX+30] is storing the value of Process Environment Block in the register EAX . It then returns the value stored in “BeingDebugged”. “Being Debugged” is a structure member of PEB, which stores the status of the process. If the status value is 0, the process is not debugged. If the status value is non zero, the process is debugged.

Instead of calling the API, for detecting debugger program, the application detects the presence of debugger by embedding the code as shown in the figure 25.0 as the API calls can be easily recognized and patched.

```
MOV EAX, FS:[0X18]
MOV EAX,[EAX+0X30]
MOV EAX,[EAX+0X68.
TEST EAX,0X70
JZ nodebugger
```

Figure 25.0 code showing embedding of Instructions

NtGlobalFlag present inside PEB, can also be used to detect debugger. The figure shown in figure 26.0 displays the code which can be used to detect debugger using NtGlobalFlag.

```
MOV EAX, FS:[0X18]
MOV EAX,[EAX+0X30]
MOV EAX,[EAX+0X68.
TEST EAX,0X70
JZ nodebugger
```

Figure 26.0 Code using NtGlobalFlag.

As shown in the figure 26.0, anti global present at offset 0x68 in process environment block is accessed, checked and compared with the value 0x70. If the value is equal to 0x70, then debugger is present else the program is executing directly. LdrInitializeThunk is the loader initialization routine that is executed first on the execution of the process. The function checks for GlobalFlag settings and then sets PEB→ NtGlobal field accordingly. The function LdrInitializeThunk present in ntdll needs to be examined. If there is no defined values for "GlobalFlag" under the registry Image File Execution Options (HKEY\Software\Microsoft\WindowsNT\CurrentVersion\IMAGE File Execution option) and if the flag PEB→ BeingDebugged is set, then PEB→NtGlobalFlag will be filled with the flag value 0x70. The flag value 0x70 is OR of

```
#define FLG_HEAP_ENABLE_TAIL_CHECK 0x00000010
#define FLG_HEAP_ENABLE_FREE_CHECK 0x00000020
#define FLG_HEAP_VALIDATE_PARAMETERS 0x00000040
```

The presence of debugger can also be checked by CPU cycles. Several CPU cycles are spent by the debugger event handling. Hence, a check on CPU cycle can be used to find out the presence of debugger. If the number of CPU cycles is more than the normal execution then it means that the application has began to debug. In order to find out the CPU cycles, x86 instruction RTDSC can be used. RTDC stands for Read Time Stamp Counter. The output of the instruction is a 64-bit value in registers EDX:EAX represents the count of ticks from the processor reset. The code shown in the figure 26.0 can be used to detect debugger.

```

Rdtsc
Push eax
Push edx
.
.
...
... ; application code
...
rdtsc
pop ebx
pop ecx
cmp edx,ebx,
ja debugger_detected

sub eax,ecx
cmp eax,0x250      ; 0x250 is the delta value
ja debugger_detected

```

Figure 27.0 Code used to detect debugger

In the code shown in the figure 26.0, it is assumed that the normal CPU cycle spent for the execution between the two consecutive RTDC instructions will be less than 0x250. The presence of debugger can also be checked by verifying the access token privilege. For any process “setdebugprivilege” is disabled. However, if any debugger loads the application then it gets enabled. When the debugger has this privilege enabled in its security token, it inherits the security token of the debugger at the time of loading any process into debugger. This enables the “setdebugprivilege” for the application.

The access token privilege check can be done by the application by performing successful operation with the help of debug privilege flags. One of the operations is to open a process like CSRSS.EXE, which permits access only to the system process . If the application is able to open these processes correctly then it can be inferred that the debug privilege in the application is enabled.

### 5.3.1 Breakpoints

For debugging purposes break points are used. Break points causes intentional pausing of program. Break point can be classified into two parts.

#### 5.3.1.1. Software breakpoint

In order to apply software break point the byte present at a particular location is modified and replaced with 0xCC. This will result in the generation of interrupt 3 i.e. breakpoint interrupt. Debugger debugging the program catches this exception and replaces the original byte at the location (where 0xCC is inserted) before passing the control back to the application.

### 5.3.1.2 Hardware breakpoint

Debugger uses CPU debug register in order to apply hardware break points. There are 8 debug registers present in the system ranging from DR0-DR7. The processor uses only 6 debug registers in order to control the debug feature. These registers can be accessed by the variable of MOV instruction; however, these instructions have to be executed at privilege zero.

These registers store the linear address of the breakpoint. The stored linear address can be the same as the physical addresses or it needs to be translated to the physical addresses. The translation is required when the paging is enabled.

The break point conditions are further determined with the help of debug register DR7. It also determines selective enabling and disabling of these conditions. For each debug address register DR0-DR3 there is a corresponding R/W0 to R/W4 field. These fields are of two-bit size. These two bits determine the type of action, which will result in termination of the execution. The description of these bits is as follows.

- 00 → Break on Execution
- 01 → Break on write
- 10 → Not Defined.
- 11 → Break on data read or write

Similarly, Len0 to Len4 fields of size two bits are associated with the corresponding debug address register DR0-DR3.

Its meaning can be interpreted as follows.

- 00 → one byte length
- 01 → two-byte length
- 10 → undefined
- 11 → four-byte lengths

If the break point condition is set as break on execution and the length bits contain values other than 00, then it will be interpreted as invalid condition. As shown in the figure, bits L0 to L3 and G0 to G3 indicate selective enabling of the corresponding four debug addresses at D0-D3. If local enabling is set it means that the

breakpoint is applicable for a particular task. However, if global enabling is set then it signifies that the condition is enabled for all the tasks. The local enable bits get reset after every task switching, in order to avoid unwanted break point condition for other tasks. In order to apply breakpoint condition to all tasks, global enabling flag is used. The significance of LE and GE bits are used to control “exact data breakpoint match” feature of the processor. If any of these bits are set then processor slows down the execution so that data

breakpoint is reported on the instruction that causes them. DR6 is the debug status register which is accessed by the debugger to determine the debug condition that has occurred. When the processor detects an enabled debug exception, it sets the low-order bits of this register (0,1,2,3) before entering the debug exception handler.

The software break point can be detected as follows. To rehash, the software break point is applied by replacing the byte with 0xCC (interrupt 3). So, the software break point can be detected by searching for the presence of 0xCC at the start of instruction.

Calculating the check sum of the protected block and comparing it with the original checksum can also detect software break point. After applying software break point, the bytes are replaced with 0xCC. Hence, the new checksum will be different from the old one.

### 5.3.1.3 Detecting Hardware Breakpoint

Detection of hardware breakpoint requires the debug register to check the values stored inside them. The debug registers cannot be accessed in ring3. In order to access the value of Debug register, structure exception handler can be used;

it reads the value stored inside the debug register. Using the Context structure, which is passed as a parameter to the exception handler, can check the value of debug register. The code shown in the figure 27.0 shows the detection of hardware break point.

```

Push exception_handler // Inserting handler in SEH chain
Push dword [fs:0]
Mov [fs:0],esp

And eax, 0           // Instruction causing exception
mov [eax],0

pop dword [fs:0]    // Removing handler from SEH chain
add esp,4

;;; Application code

exception_handler
mov eax,[esp+0x0C]   ; accessing third parameter which pointer to CONTEXT
structure

cmp dword [eax+0x04],0
jne Breakpoint_found
jmp Continue_execution

Breakpoint_found
; Perform the desired operation after detection of break point

Continue_execution
Add [eax+0xb8],9    ; Handing exception by incrementing eip by 9
ret

```

Figure 28.0 showing the detection of hardware breakpoint.

## 5.4 Virtual Machine Detection

Use of VmWare for the malware analysis is discussed in the chapter of Malware. The use of Virtual machine provides the following functionality.

- Multiple Operating system
- Snap shot of the machine state
- Easy To Monitor

Since security researcher use Virtual Machine for the analysis of malwares, the malware writer generally adds some checks to detect the presence of virtual machine environment. In the next subsection, detection of VM is discussed.

### 5.4.1 Checking fingerprint inside memory, file system and registry

The run time environment of the virtual machine contains signatures like VMware, vmx. in the process/services, filename and registry entry. This can be searched to detect the presence of virtual machine. This technique can be evaded through hooking the system calls and manipulating its input and output.

### 5.4.2 Checking system tables

Vmwares can be detected by putting checks on the pointers, which point to the kernel data structure like Interrupt Descriptor Table (IDT), Global Descriptor Table (GDT) and Local Descriptor Table (LDT). This technique provides the inherent advantage to almost all kind of Virtual machines, Windows and Linux operating system. It might be difficult to evade, as this technique is an integral part of the virtual machine environment.

One of the common examples for the implementation of the system table check (for the detection of virtual machine) is redpill, developed by Joanna Rutkowska. Redpill uses an instruction called SIDT (Store Interrupt Descriptor Table).SIDT stores the address of interrupt Descriptor table pointer in the memory through the register Interrupt Descriptor Table Register (IDTR). The logic inside this tool is based on the fact that the location of IDT in the host machine is far lesser than the location of IDT in the guest machine. The code compares the location of IDT with the constant 0xD0000000 and if the value is found to be less than 0xD0000000 then the tool will assume that it

is running on the host machine else it will give a message that it is running on the Virtual machine. This tool is found to be very acute in Linux as well as Windows Operating System. The code shown in figure 29.0 demonstrate the approach.

```
int swallow_redpill() {
    unsigned char m[2+4],
    rpill[] = "\x0f\x01\x0d\x00\x00\x00\x00\xc3";
    *((unsigned*)&rpill[3]) = (unsigned)m;
    ((void(*)())&rpill)();
    return (m[5]>0xd0) ? 1 : 0;
}
```

Figure 29.0 showing the code of redpill.

Extending the logic of Red Pill, Tobias Klein has written a new tool Scoopy Suite, which apart from checking IDT also checks the location of GDT, LDT using SIDT, SGDT, and SLDT processor instructions. The logic inside the tool is same as that of the red pill. Rather than depending only on IDT, this tool counts on other structures as well.

In Windows Operating System, it checks the starting byte of the pointer to IDT structure; if the starting byte starts with 0x80 then it will assume that it is running in the host system. In Linux, it checks the value with 0xc0. Then the code checks for the logic, to compare the pointer in GDT's location with 0xc0XXXXXX. If the pointer of LDT is located at 0x0000, then it means that it is running in the real system else it is running in the Virtual PC.

#### 5.4.3 Checking processor instruction set

Processor instructions can be used to detect virtual machine. There are few non-standard x86 instructions that are used by Virtual PC for guest-to-host communication. However, execution of these non-standard instructions in host PC would result in processor exception or error. Operating system will search for the exception handler to handle the exception or will terminate the program. These instructions can be used by the application to determine if the application is running in Virtual PC or in the host system. The tool like VMdetect uses the process instruction set check, to detect Virtual PC.

The code shown in the figure 30.0 can be used to detect VMWare. The IsInsideVMWare() will return True if it is a Virtual Machine else it will return False.

```
bool IsInsideVMWare()
{
    bool rc = true;

    __try
    {
        __asm
        {
            push  edx
            push  ecx
            push  ebx

            mov   eax, 'VMXh'
            mov   ebx, 0 // any value but not the MAGIC VALUE
            mov   ecx, 10 // get VMWare version
            mov   edx, 'VX' // port number

            in    eax, dx // read port
                // on return EAX returns the VERSION
            cmp   ebx, 'VMXh' // is it a reply from VMWare?
            setz [rc] // set return value

            pop   ebx
            pop   ecx
            pop   edx
        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        rc = false;
    }
    return rc;
}
```

Figure 30.0 Code for detection of VmWare.

The magic number 0x564D5868 (in ASCII 'VMXh') is loaded in the EAX register. The parameter of the command that has to be sent is loaded in EBX register. The command is loaded in the ECX register. For example, in the figure 29.0, the command 0x0A, is loaded. This command returns the version number of VMWare through the port 'VX'. After the execution, if 'VMXh' is present in the EBX register, then it can be inferred that VMWare is being used.

## 5.5 Unpacking

Executable packing is carried out to compress and/or encode the original code and data present in the executable. At the time of execution, the original data/code is decoded back and the execution control is passed to it. Packing process does not affect the functional behavior of any application. Hence it is difficult for the normal user to identify it. One of the main usage of packing an executable is to prevent reverse engineering or to obfuscate the content of

the executable. Although it cannot prevent reverse engineering, it can make reverse engineering more tedious. Loading the packed executable in any disassembler will generate invalid set of instructions. Hence it is required to unpack the executable for a better and effective analysis of a packed executable. The following section discusses the process of unpacking of any window binary

### 5.5.1 Manual Unpacking of Software

The process of manual unpacking can be classified into 3 steps. The first step involves finding the original entry point of an executable, second step involves taking process dump and the third involves fixing entries in import address table. After performing the first two steps, static analysis on the code can be performed. After performing the third step, dynamic analysis can be performed.

#### 5.5.1.1 Finding an Original Entry Point of an Executable

Before explaining the intricacies to find original entry point, few of the basic concepts about packing process, will be discussed. To protect application from getting reverse, packer encodes/encrypts the original application so that when opened in any disassembler / debugger, it will not show the correct or the original sequence of instructions. However, at the time of execution the encrypted code has to be decoded or decrypted back to interpret the original executable properly. So to achieve the objective of decoding the binary correctly, packers add some instructions in the packed executable, in order to unpack the encoded/encrypted executable. The instructions added by the packer perform decoding/decryption process. The process of decryption is performed in memory at the time of execution, restoring the state of the application. Packer works on any standard and precompiled executable. Hence, the unpacking module has to be independent of the original application. Packer works on any standard and precompiled executable. Hence, the unpacking module has to be independent of the original application.

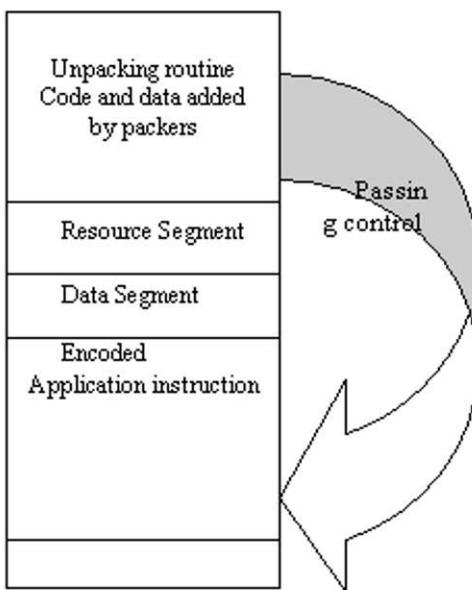


Figure 31.0 showing the instruction for Unpacking code module.

Packers can add independent instructions (unpacking module) in the encoded executable. These instructions perform the task of decrypting the encoded executables. It can either be added after encode/encrypted executable image or before encode/encrypted executable image. Almost all the packers, add instruction after encode/encrypted executable image (means at higher virtual offset). The second approach of adding instructions is to decoded or decrypt executable before the executable image that is not feasible. In a 32 bit windows all executable get mapped/loaded at base address 0x00400000. At the time of linking (after compilation) the linker links an application by assuming that application will get loaded at a base address of 0x00400000. If the instruction to decode is added before the executable, then the virtual address of the original image base of an application is no longer 0x00400000, as the original resolved virtual address by the linker will be different from the currently loaded executable. This will result in an abnormal behavior of an executable.

```

.text:00401022          mov    edx, [ebp+var_8]
.text:00401025          jmp    ds:off_401054[edx*4]
.text:0040102C
.text:0040102C loc_40102C: mov    eax, [ebp+arg_0] ; DATA XREF: .text:off_401054$o
.text:0040102F          add    eax, 1
.text:00401032          mov    [ebp+var_4], eax
.text:00401035
.text:00401035 loc_401035: mov    ecx, [ebp+arg_0] ; CODE XREF: sub_401000+25$j
.text:00401035           add    ecx, 1 ; DATA XREF: .text:00401058$o
.text:00401035           mov    [ebp+var_4], ecx
.text:0040103E
.text:0040103E loc_40103E: mov    edx, [ebp+arg_0] ; CODE XREF: sub_401000+25$j
.text:0040103E           add    edx, 1 ; DATA XREF: .text:0040105C$o
.text:0040103E           mov    [ebp+var_4], edx
.text:00401047
.text:00401047 loc_401047: mov    eax, [ebp+arg_0] ; CODE XREF: sub_401000+25$j
.text:00401047           add    eax, 1 ; DATA XREF: .text:00401060$o
.text:00401047           mov    [ebp+var_4], eax
.text:0040104D
.text:0040104D loc_40104D: mov    esp, ebp ; CODE XREF: sub_401000+20$j
.text:0040104D           pop    ebp
.text:00401053
.text:00401053 sub_401000: retn
.text:00401053
.text:00401053 loc_401050: endp
.text:00401050
.text:00401050 loc_401054: dd offset loc_40102C ; DATA XREF: sub_401000+25$o
.text:00401058          dd offset loc_401035 ; DATA XREF: sub_401000+25$o
.text:0040105C          dd offset loc_40103E ; DATA XREF: sub_401000+25$o
.text:00401060          dd offset loc_401047 ; DATA XREF: sub_401000+25$o

```

Figure 32.0 Figure for unpacking instructions.

As shown in the figure 32.0 the instruction

```
.text:00401025      jmp    ds:off_401054[edx*4]
```

at address 00401025 is referring to a memory location 00401054. The linker has resolved these address references with references to image base, which is 0x00400000. If a packet adds any bytes before the image of original application, then the original code of an application will get mapped to the base address greater than 0x040000 ( $\geq 0x00400000 + \text{size of the byte added}$ ). At the time of execution, address resolved by the linker will result in an abnormal crash. Hence, all the address references should be done with reference to the new image base. This in turn will add to the complexity of address references. So, the packets prefer to load the unpacking instructions after the image of original applications, which will be used to find the Original Entry Point (OEP).

The unpacking instructions are present after the encoded executable image. However, it gets the control to decode the encoded bytes, post which, it transfers the control to decode the original instruction

To find the original entry point, first load the application in any debugger and turn on the tracing options. This will log all executed instructions with their corresponding address. When the program is executed the debugger will log all executed instructions and will update the trace log accordingly. Terminate the

process, when the original application gets executed. During the analysis of trace logs, it can be inferred that the higher address instructions will get executed first. To find the instruction, which transfers control from higher address to lower address, locate two consecutive instructions such that the address of first instruction is high with reference to second instruction. As discussed above, the unpacking module located at higher address passes control to the decode application at lower address, where the lower address is the OEP.

Another approach for finding OEP is by using Ollydbg plugin in OllyBone developed by Joe Stewart. Details of this plugin are available at <http://www.joestewart.org/ollybone>. The plugin uses the concept of split TLB. The Intel processor to protect memory pages from execution while allowing read/write access uses TLB. TLB is a cache buffer used for virtual – physical address translation. It is used to enhance the performance of the system by providing information without performing an expensive page table walk operation for memory access. Whenever the CPU wants to access the given virtual address, it will first check if the TLB has a cached translation. If the address is found on the TLB it will take the physical address from TLB, otherwise it will perform a page table look up for the required address translation.

Intel from Pentium architecture has started providing split TLB architecture. In split TLB architecture, virtual/physical translations are cached into two independent TLBs depending on the access type. Virtual / physical translations are cached into two independent TLB's. This depends upon the access type. Instruction fetch related memory access will load the ITLB and update the DTLB for data access. OllyBone comprises of a windows kernel driver that implements the page protection for arbitrary memory page access. It also consists of an OllyDbg plugin that can be used to communicate with the driver. If a protected page is accessed by the CPU for execution, it will result in calling of INT1 handler, which in turn will return the control to OllyDbg. The following steps have to be followed to use this plug-in for finding out OEP.

- Load the packed program.
- Find out which section in the memory map will be executing when the unpacking is finished. Most probably, this section will be the first section seen through PE Editor.
- Set break-on-execute flag for the section, which will load the kernel driver into memory and protect the desired physical memory pages from being executed.
- Run the program.
- OllyDbg will break when CPU tries to execute the first instruction in the selected section. The instruction at which OllyDbg breaks will be the original entry point (OEP).

Some packers not only add unpacking code in the higher address of the newly created segment but also make use of free space of the section in original application, where OEP resides. To explain it further, the figure 32.0 shows the original application before packing.

Section Table Viewer					
Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	00003D33	00001000	00004000	00001000	60000020
.rdata	0000087C	00005000	00001000	00005000	40000040
.data	00002148	00006000	00001000	00006000	C0000040

Do a right mouse click on a sectionname for more options...

Figure 33.0 showing the original application before unpacking

Section Table Viewer					
Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
.text	00004000	00001000	00000000	00000400	E0000020
i52.z3v0	00001000	00005000	00000000	00000400	E0000060
.data	00002000	00006000	00000000	00000400	C0000040
2bf198fd	00001000	00008000	00000000	00000400	C0000040
276xc6nv	0000D000	00009000	00000000	00000400	E0000020
xs.al3yc	00013000	00016000	00012318	00000400	E0000060
vm5zwk7j	00001000	00029000	00001000	00012800	40000080

Do a right mouse click on a sectionname for more options...

Figure 34.0 shows the original application after unpacking

The last four segments are added by the packer after encoding the original executable image. Before packing, the virtual size of the .text segment was 0x3d33 and after packing , the packing routines has increased to a size of 0x4000. 0x4000 is the maximum value that a packer can specify. The virtual size of the section is 0x4000 and the virtual offset is 0x1000. If the virtual size is more than 0x4000, then it will result in overlapping with the next section. (The next section starts from 0xt5000).

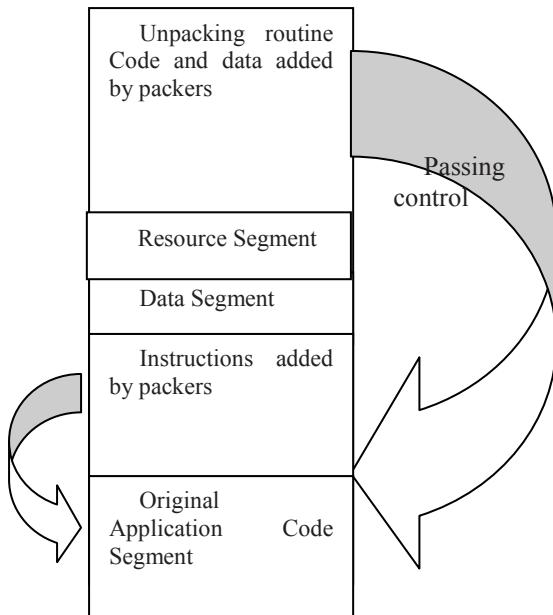


Figure 35.0 showing the working of packers

As shown in the figure 35.0 the packers add some code in the void space available in the first section. These unpacking modules are located at different segments and transfers control to the OEP. If the packing modules are located in the void space, then it will be difficult to find the OEP by using the methods, which makes use of plugins like OllyBonE.

Besides the above-mentioned methods there are some other methods, which can be used to find out the original entry point. This technique drives its strength from the fact that at the time of linking, the executable linker prep ends a start module at the entry point of an application. This module is executed first before executing the main program and it sets the environment ready for execution. Since the linker adds similar code to all the application, one of the techniques for finding the OEP is to locate for the signature of startup module. Few plugins like generic OEP finder of Peid, stores the signature of various startup modules of different linkers. Using the plugins like generic OEP finder can use these signatures to find OEP.

In order to locate OEP manually, break points can be applied on the following API's

- GetVersion
- GetVersionExA
- GetEnvironmentVariable
- LoadLibrary
- GetProcAddress
- VirtualAlloc

These APIs are called by the startup routines in order to set up the execution environment. By applying break point, API's will become closer to OEP. As the startup module calls these API's, and the address of the startup module is OEP, the start of the function needs to be determined. The start of the function can be recognized by the following code.

```
Push ebp
Mov ebp,esp
```

These two instructions are used to create a stack frame; local variables can be accessed through ebp register. However, some compilers that perform optimization might not use ebp register for creating stack frame instead use a ebp as a general purpose register. So these instructions may not be at the start of the program.

### 5.5.1.2 Taking Memory Dump

After locating the original entry point the next step is to get the memory dump of a process. Memory dump of the process, which will result in the original executable, is done after the completion of unpacking process. To get the memory dump the application will have to be looped at the OEP. Following steps can be used to get the memory dump.

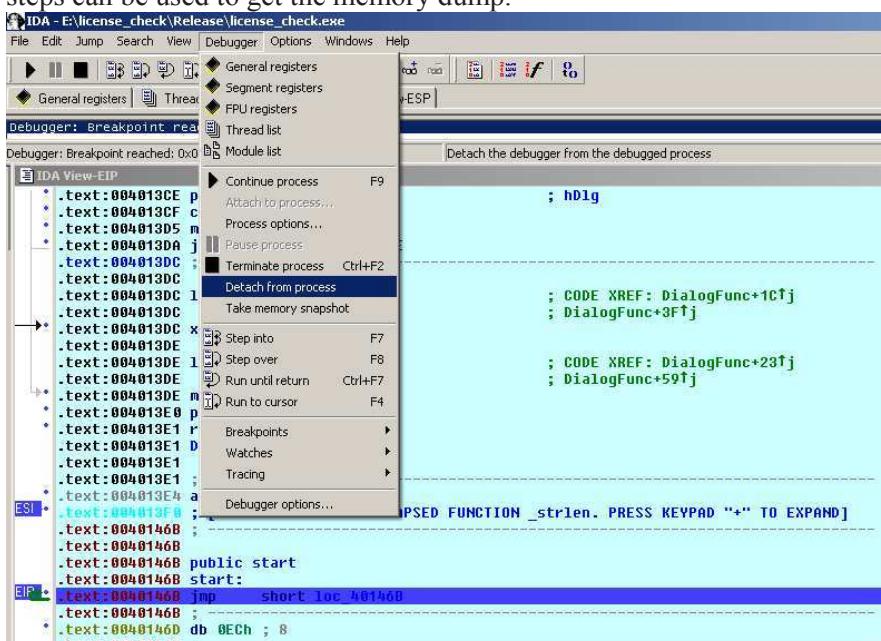


Figure 36.0 showing the memory dump.

- Apply hardware break point at the OEP.

In case of software break point, debugger puts byte 0xCC at the location where the breakpoint is applied. This results in interrupt 3, post which, the debugger takes the control and replaces 0xcc with original bytes. Since the unpacking module gets control first (before control reaches to OEP) the unpacking instruction will treat 0xcc as an encode byte and will try to decode it, which will result in generation of junk code and the application will crash. In case of hardware break point, the information is stored in the debug register and no code change is required.

- Executing the Application. The Application will stop at OEP.
- Change the instruction and make it loop back to itself and then detach it from debugger.
- Next step will be to take the memory dump. Proc dump (available at) can be used to take the dump of process.



Figure 37.0 showing proc dump

- Proc dump reads the information stored in the PE header to find out segment information like virtual size and virtual address. It then dumps the segments and file as it is in the hard disk. Now the size in file address and RVA size has to be the same. But the section information in the original application contains different virtual and raw address. So the dump needs to be fixed



Figure 38.0 Showing Information from PE header

- The next step involves the changing of the entry point of the PE header to the original entry point using PE editor.
- Open the application in a hex editor like “cygus hex editor” (available for download). Go to the entry point of the application and restore the original byte, which was present before replacing it with jmp loop back bytes.

The memory dump from original application, which can be obtained from the above-mentioned steps, can be loaded in IDA pro for static analysis.

#### 5.5.1.3 Import Table Reconstruction

Dynamic analysis requires reconstruction of import symbols and import data structures. Imported functions, are certain functions in the caller module, which doesn't have the code in the executable. The executable stores only certain information about these functions; loader use this information at load time and stores the addresses of the functions in the executable enabling the caller modules to call the functions using these addresses.

For storing these imports information and address, certain well-defined data structures should be known in order to reconstruct them. The following code represents the NTHHeader present inside the PEheader.

```
Struct IMAGE_NT_HEADERS STRUCTURE {
    Signature dd ?
    IMAGE_FILE_HEADER FileHeader
```

```
    IMAGE_OPTIONAL_HEADER optionalHeader
}
```

Data directory is the last member of IMAGE\_OPTIONAL\_HEADER. Data directory is an array 16 IMAGE\_DATA\_DIRECTORY structure. Following figure will explain about these structures as discussed above.

Each member of the data directory is a structure called IMAGE\_DATA\_DIRECTORY, which has the following definition: -

```
Struct IMAGE_DATA_DIRECTORY STRUCT {
    VirtualAddress dd ?
    ISize dd ?
}
```

Second entry comprises of Import Symbols. So the **virtual Address** field will contain the address of IMAGE\_IMPORT\_DESCRIPTOR array and **size** will contains the size in byte of the data structure pointed by virtual address.

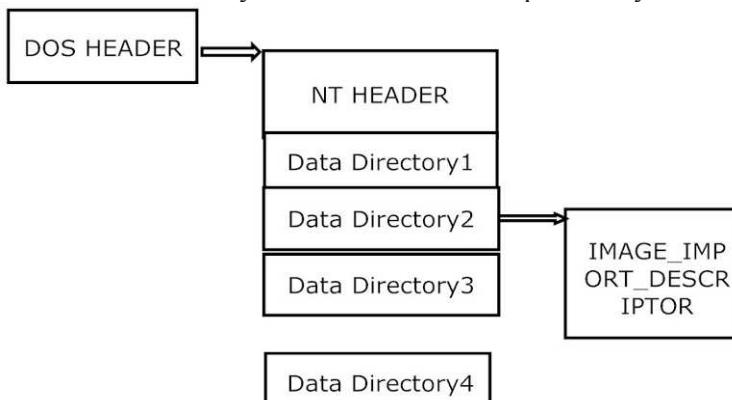


Figure 39.0 showing the import table structure

IMAGE\_IMPORT\_DESCRIPTOR is a data structure, which stores information about import symbols. There is one IMAGE\_IMPORT\_DESCRIPTOR for each imported executable. The end of the IMAGE\_IMPORT\_DESCRIPTOR array is indicated by an entry with fields all set to 0.

The structure of IMAGE\_IMPORT\_DESCRIPTOR is as follows: -

```
Struct IMAGE_IMPORT_DESCRIPTOR STRUCT {
    OriginalFirstThunk dd ?
    TimeDateStamp dd ?
    ForwarderChain dd ?
```

```
Name1 dd ?
FirstThunk dd ?
}
```

The screenshot shows the PEView interface. On the left, there is a tree view of the file structure for 'license\_check.exe'. It includes sections like IMAGE\_DOS\_HEADER, IMAGE\_NT\_HEADERS, and various sections (.text, .rsrc, .data, .rdata). Under .rdata, there is a group for 'IMPORT Address Table' containing 'IMPORT Directory Table', 'IMPORT Name Table', and 'IMPORT Hints/Names & DLL Names'. On the right, there is a table of imports:

RVA	Data	Description	Value
00018454	00018524	Import Name Table RVA	
00018458	00000000	Time Date Stamp	
0001845C	00000000	Forwarder Chain	
00018460	000186D2	Name RVA	USER32.dll
00018464	00018094	Import Address Table RVA	
00018468	00018490	Import Name Table RVA	
0001846C	00000000	Time Date Stamp	
00018470	00000000	Forwarder Chain	
00018474	0001894E	Name RVA	KERNEL32.dll
00018478	00018000	Import Address Table RVA	
0001847C	00000000		
00018480	00000000		
00018484	00000000		
00018488	00000000		
0001848C	00000000		

Figure 40.0 Shows the organization of IMAGE\_IMPORT\_DESCRIPTOR using PEView.

IMAGE\_IMPORT\_DESCRIPTOR comprises of many structures, which are discussed below.

- **OriginalFirstThunk:** - This member contains the RVA (pointer) of an array of IMAGE\_TUNK\_DATA structures. IMAGE\_TUNK\_DATA structures, is a union of dword size. This can be considered as a pointer to IMAGE\_IMPORT\_BY\_NAME structure. The structure of IMAGE\_IMPORT\_BY\_NAME structure is as follows: -

IMAGE\_IMPORT\_BY\_NAME STRUCT

Hint

Name1

IMAGE\_IMPORT\_BY\_NAME ENDS

It contains the index of the export table of the DLL. The loader uses this field so that it can look up for the function in the DLL's export table quickly. This value is not mandatory and in some cases you will find that the linker will set its value to 0.

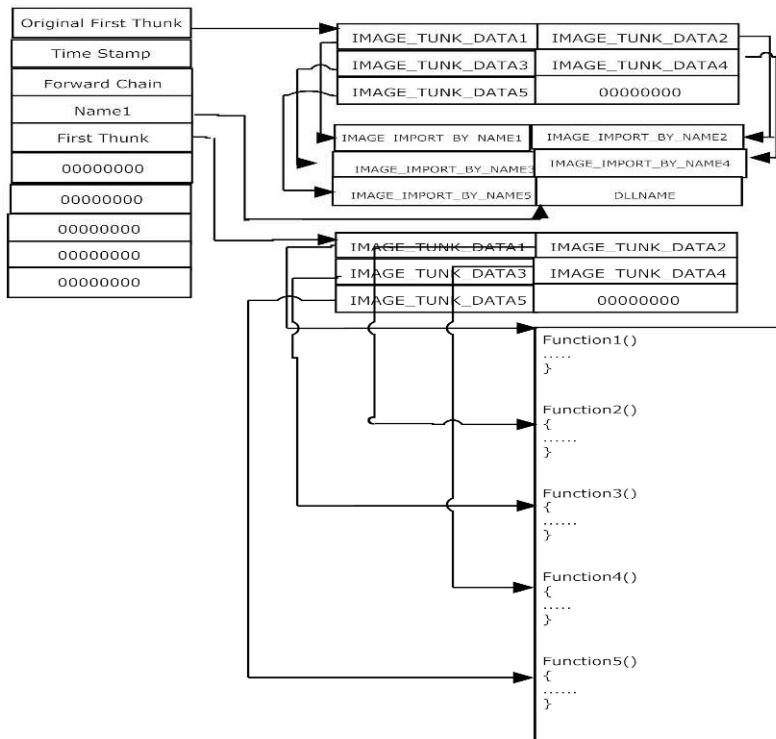


Figure 41.0 Showing the organization of import data structure.

- i. **Time/Date Stamp:** - After the image is bound, this field is set to the time/date stamp of the DLL. This field is not mandatory; it can be zero.
- ii. **Forwarder Chain:** - The index of the first forwarder reference. This field is not mandatory; it can be zero.
- iii. **Name:** - This member contains the RVA (pointer) of an ASCII string that contains the name of the DLL.
- iv. **FirstThunk:** - As the name suggests the FirstThunk is very similar to that of OriginalFirstThunk . Similar to FirstThunk it also contains pointer (RVA) to array of IMAGE\_THUNK\_DATA structures. Although both the arrays contain same value, they are at different locations in the executable.

To reconstruct the import, table name of the entire imported API is required. One of the ways is to search for the name of the imported APIs in the dump executable. However, the unpacking module may distort the name of few or all API after using / loading the address, so the method of getting the list is not reliable. Another method of knowing the name of the API is by

using tools like Re-Virgin (available at [http://www.re-virgin.com](#)) or Imp-REC(available at [http://www.impactsoft.com](#)). These tools require the address and the length of the IAT (Import Address Table as discussed earlier array of IMAGE\_THUNK\_DATA structures, it is a location where loader loads addresses of the imported functions). Hence the location and the length of IAT is required. To rehash, IAT stores the address of the API. In order to call the API, the application must refer IAT and the call will be redirected to IAT.

In order to find out the address of the import table, analysis of the code of unpacking application is required. This will give the indirect call referring memory location, which stores the address of the function. This resolves to determine the instruction template **CALL DWORD PTR [XXXXXX]**. Here XXXX can be the address or name of the API, as few disassemblers resolved the name of the API. As shown in figure 41.0 the highlighted instruction, which is calling the function indirectly by referring the memory location needs to be determined.

0040146E	6A FF	PUSH -1
00401470	68 F0804100	PUSH license_.004180F0
00401475	68 A01F4000	PUSH license_.00401FA0
0040147A	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
00401480	50	PUSH EAX
00401481	64:8925 00000000	MOV DWORD PTR FS:[0],ESP
00401488	83EC 58	SUB ESP,58
0040148B	53	PUSH EBX
0040148C	56	PUSH ESI
0040148D	57	PUSH EDI
0040148E	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
00401491	FF15 600004100	CALL DWORD PTR DS:<KERNEL32.GetVersion>
00401497	33D2	XOR EDX,EDX
00401499	8AD4	MOU DL,AH
0040149B	8915 80954100	MOV DWORD PTR DS:[419580],EDX
004014A1	8BC8	MOV ECX,EAX
004014A8	31E1 FF000000	MUL ECX,ECX

Figure 42.0 shows the function

Once the instruction template **CALL DWORD PTR [XXXXXX]** is located , the next step involves reading the address [XXXXXX], or 0x40801C and jumping to that address. This address will be in the IAT.

Figure 43.0 shows the memory location by using memory view of Olydbg.

The figure 43.0 figure shows the memory organization at the 0x40801C desired location. Since the last byte in every DWORD entry ends with 7C, indicating the address of function of Kernel32.dll, it can be concluded that this memory is residing inside IAT. Generally the byte in the import tables will be in an ordered way.

The reason is that, it stores the addresses of API for a dll. The first byte of the address will be similar. For example, in the enclosed figure, the first byte of the instruction is 7C. So it can be concluded that the import table will start from the address xxxx and end at the address xxxxx

By providing the inputs (start address of IAT and the length of IAT) to Re-Virgin, the name of the API used by the original program can be listed. The figure 44.0 shows the output of the Re-Virgin tool after providing it with the address range of IAT.

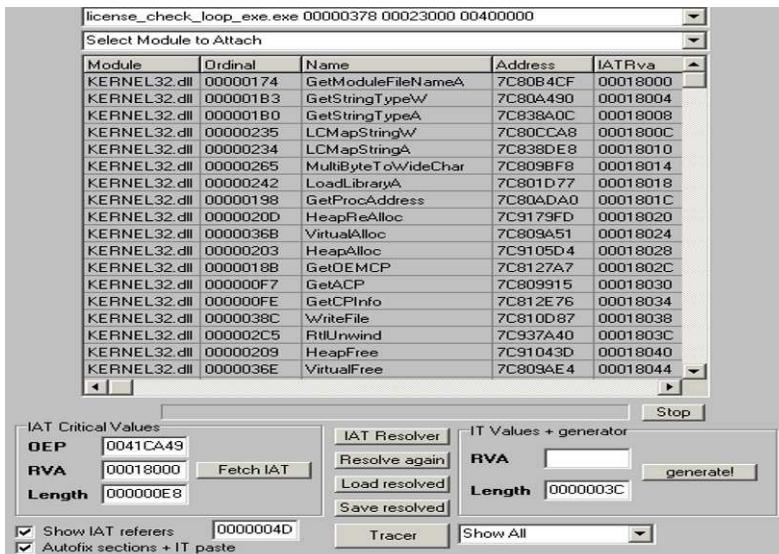


Figure 44.0 showing the output of Re-Virgin Tool.

Although the name of API can be determined with the help of above mentioned procedure, the techniques like, import redirection, code emulation the unpacking process can be made a challenging task.

#### 5.5.1.4 Import redirection and Code emulation

ImpREC functionality can be extended and customized for a particular packer through plugins that help to find out the name of the API's. The following link will explain how to write plugins for Import REC. After determining the name of all the API's used by the application, the next step will be to structure the import name table such that the loader is able to load the executable.

#### Manual Import Name Table Reconstruction

There can be two approaches to construct import address. The first approach can be termed as

- Top to bottom approach. For this  
IMPORT\_DESCRIPTOR structure entry then  
IMAGE\_TUNK\_DATA structure and then following it  
IMAGE\_IMPORT\_BY\_NAME structure is constructed.
- Bottom to Top Approach.  
IMAGE\_IMPORT\_BY\_NAME then  
IMAGE\_TRUNK\_DATA and then IMPORT\_DESCRIPTOR

Here, the import table reconstruction will be done by using the second approach.

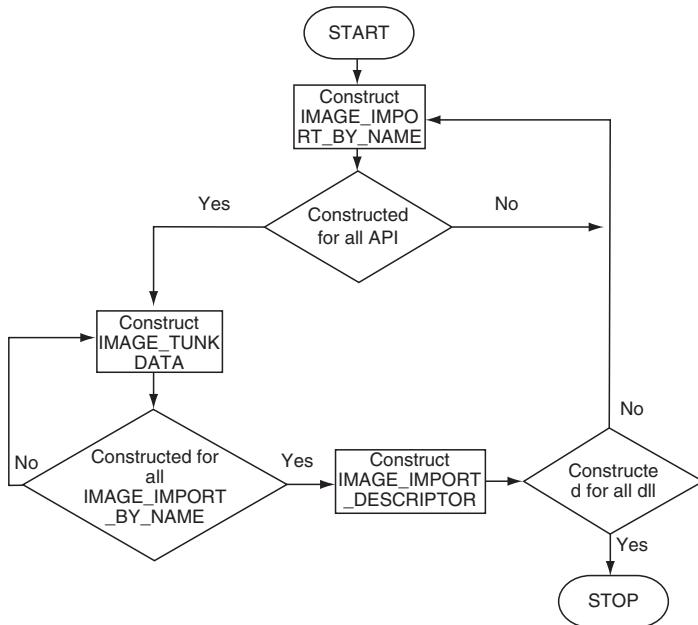


Figure 45.0 Shows import table reconstruction.

The first step requires the construction of IMPORT\_DESCRIPTOR. The structure of IMPORT\_DESCRIPTOR will be as follows

```

IMAGE_IMPORT_BY_NAME STRUCT
  Hint dw ?
  Name1 db ?
IMAGE_IMPORT_BY_NAME ENDS
  
```

Since the “hint” field is not essential, its value can be set to 0. Since the name of all API’s used by the steps mentioned in previous section know the application, the free space is needed in the exe to construct the import name table. The free space should be long enough such that all the API entries can be adjusted.

00460C0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
00460D0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
00460E0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
00460F0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
0046100	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
0046110	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
0046110	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
0046120	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
0046130	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
0046140	00 00 43 72 65 61 74 65-46 69 6C 65 41 00 00 00	.CreateFileA...
0046150	00 00 4C 6F 61 64 49 63-6F 6E 41 00 00 00 00 00	.LoadIconA...
0046160	00 00 47 65 74 55 73 65-72 4E 61 6D 65 41 00 00	.GetUserNameA...
0046170	00 00 44 6C 6C 52 65 67-69 73 74 65 72 53 65 72	.DllRegisterSer...
0046180	76 65 72 00 00 00 00 00-00 00 00 00 00 00 00 00	ver...
0046190	00 00 77 73 63 61 6E 66-00 00 00 00 00 00 00 00	.wscanf...
00461A0	00 00 62 69 6E 64 00 00-00 00 00 00 00 00 00 00	.bind...
00461B0	00 00 77 63 74 6F 62 00-00 00 00 00 00 00 00 00	.wctob...
00461C0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
00461D0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.
00461E0	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.

Figure 46.0 shows the import table reconstruction

Figure 46.0 shows the reconstructed IMAGE\_IMPORT\_BY\_NAME for the all API's referring to the list of API names. The above-mentioned figure shows the entry of IMAGE\_IMPORT\_BY\_NAME for each API containing the value in Hint as 0, and the name of the API. The next step requires the reconstruction of IMAGE\_TUNK\_DATA structure. The IMAGE\_TRUNK\_DATA is a structure which contains DWORD (pointer to IMAGE\_IMPORT\_BY\_NAME entry). Construction of IMAGE\_TUNK\_DATA data structure pointing to each IMAGE\_IMPORT\_BY\_NAME entry will be required. For this, the free space inside the executable will be required. The free space should be greater than or equal to (Total number of API +1) \* size of (DWORD), so that the entry for all the API calls will be available. The last entry will contain the value Zero indicating the end of the entry of IMAGE\_TUNK\_DATA structure.

Figure 47.0 showing the reconstruction of IMAGE\_TRUNK\_DATA

Reconstruction of IMAGE\_TUNK\_DATA is explained in the figure 47.0. As shown in the figure, the IMAGE\_IMPORT\_BY\_NAME entry for the API, Dispatch MessageA is at the address 0x0000497E. The final entry will contain value zero, indicating the termination of the array of this structure. In the figure 47.0, it can be seen that the value 7E 49 00 00 stored at address 0x00004700 is denoting the address 0x0000497E.

The similar steps will have to be followed for all the API's. After which the final step requires the construction of IMPORT\_DESCRIPTOR. The structure of IMPORT\_DESCRIPTOR is as follows.

## IMAGE IMPORT DESCRIPTOR STRUCT

## OriginalFirstThunk dd ?

TimeDateStamp dd ?

ForwarderChain dd ?

Name1 dd ?

FirstThunk dd ?

## IMAGE IMPORT DESCRIPTOR ENDS

**OriginalFirstThynk**: It is a dword, which will point to an array of newly, reconstructed IMAGE\_TUN\_DATA structure. For example as shown in the figure, 46.0 it will contain the value 0x00004700 at the starting RVA of the array of IMAGE\_TUNK\_DATA

**TimeDateStamp**, not a mandatory field so it can be set to zero.

**ForwardChain** not a mandatory field so it can be set to zero

**Name1** is a pointer to the name of the DLL for which reconstruction of the IMAGE\_IMPORT\_BY\_NAME has been done. In the case shown in the figure, the RVA of ASCII string of “Kernel32.dll” is stored.

**FirstThunk** is a pointer to an array IMAGE\_TUNK\_DATA structure where loader will store the address of imported functions for that particular DLL, so that applications can call these functions. Its address is used throughout the exe so it remains fixed. Hence, it cannot be changed and it will point to the array of IMAGE\_TUNK\_DATA structure, which is already discovered for finding out the name of the API.

To summarize, the reconstruction of single DLL has been done. The step has to be repeated for each DLL.

## 5.6 Conclusion

Software reverse engineering is a very handy mechanism on the binary to reveal the design of the software. In order to protect the binary from getting reversed, binary code is added with various anti reversing techniques. Anti Disassembly technique is used to confuse the disassembler. This forces the disassemblers to generate incorrect disassembled code by exploiting the implementation design of the disassemblers. Anti-disassembly techniques can successfully confuse linear and Recursive Traversal Disassemblers as well.

Apart from exploiting the implementation design there are also some other techniques like, Self code modification, dynamic encryption, decryption which encodes the instruction bytes so that the disassembler does not disassemble the instruction properly and decode it only at the time of execution. There is one more technique called as virtual machine runtime environment, which changes the standard x86 instruction byte into p-codes that can only be understood and executed by the customized runtime environment of virtual machine. Anti debugging techniques basically detect the presence of debugger environment. If the application has already started debugging, anti debugger does something really bad and exits the program. Debugging can also be detected through break points. There are two type of break points namely, software and hardware break point. Software break points can be detected by checking the integrity of the code and hardware breakpoints are detected through debug register. Virtual machine (like VM ware) provides the feature of multiple OS in a machine. There are certain techniques that detect the presence of virtual machine environment and execute the program. Nearly, all-malicious software is protected with a packer. Packing is done to protect software from reverse engineering. Packers encode

the original bytes of an application and add unpacking subroutine into the packed executable so that at the time of execution, it decodes these instructions back to its original form and then transfer the control to the original application. For analysis, packed application needs to be unpacked. Unpacking process can be divided into three steps.

1. Finding Original Entry Point OEP is the instruction address to which unpacking module transfers the control after unpacking/decoding the original application.
2. Taking memory dump and updating PE header: - The application has to be in decoded state at the time of execution so taking memory dump can retrieve the original application. Few information stored in PE header like entry point and section size need to be updated accordingly.
3. Import table reconstruction: - As packers destroy the import information after loading the module, reconstruction is needed to import the table.

After performing these steps the application is unpacked and can be further analyzed.

## APPENDIX

Hex Signature	ASCII Signature	File Description	File Extension
11 byte offset] 00		Palmpilot Database/Document File	PDB
00 00 01 00		Windows icon file	ICO
00 00 01 Bx		MPEG video file	MPEG , MPG
00 00 02 00		Windows cursor file	CUR
		QuattroPro for Windows Spreadsheet file	WB2
00 00 02 00 06 04 06 00 08 00 00 00 00 00		Lotus 1-2-3 spreadsheet (v1) file	WK1
00 00 1A 00 00 10 04 00 00 00 00 00		Lotus 1-2-3 spreadsheet (v3) file	WK3
00 00 1A 00 02 10 04 00 00 00 00 00		Lotus 1-2-3 spreadsheet (v4) file	WK4
00 00 49 49 58 50 52 <b>or</b> 00 00 4D 4D 58 50 52	..IIXP R ..MMXP R	Quark Express document	QXD
[7 byte offset] 00 00 FF FF FF FF	[7 byte offset] ..yyyy	Windows Help file	HLP
00 01 00 00 4D 53 49 53 41 4D 20 44 61 74 61 62 61 73 65	....MS IS AM Database	Microsoft Money file	MNY
00 1E 84 90 00 00 00 00		Netscape Communicator (v4) mail folder	SNM
00 5C 41 B1 FF	. \A± .	Mujahideen Secrets 2 encrypted file	ENC
[512 byte offset] 00 6E 1E F0	[512 byte offset]	PowerPoint presentation subheader	PPT

	.n. <sup>o</sup>	(MS Office)	
01 00 00 00		Extended (Enhanced) Windows Metafile Format, printer spool file	EMF
01 00 00 00 01		Unknown type picture file	PIC
01 10		Novell LANalyzer capture file	TR1
01 DA 01 01 00 03	.U....	Silicon Graphics RGB Bitmap	RGB
01 FF 02 04 03 02	.Y....	Micrografx vector graphic file	DRW
02 64 73 73	.dss	Digital Speech Standard (Olympus, Grundig, & Phillips)	
03		MapInfo Native Data Format dBASE III file	DAT
03 00 00 00 41 50 50 52	.....AP PR	Approach index file	ADX
04	.	dBASE IV data file	DB4
00 01 01		OpenFlight 3D file	FLT
00 06 15 61 00 00 00 02 00 00 04 D2 00 00 10 00		Netscape Navigator (v4) database file	DB
00 11 AF		FLIC Animation file	FLI

07		A common signature and file extension for many drawing programs	DRW
07 64 74 32 64 64 74 64	.dt2ddt d	DesignTools 2D Design file	DTD
08		dBASE IV or dBFast configuration file	DB
512 byte offset] 09 08 10 00 00 06 05 00		Excel spreadsheet subheader (MS Office)	XLS
0A nn 01 01		ZSOFT Paintbrush file	PCX
0C ED	.i	Monochrome Picture TIFF bitmap file (unconfirmed)	MP

0D 44 4F 43	.DOC	DeskMate Document file	DOC
0E 57 4B 53	.WKS	DeskMate Worksheet	WKS
[512 offset] 0F 00 E8 03	[512 byte offset] ...è.	PowerPoint presentation subheader (MS Office)	PPT
11 00 00 00 53 43 43 41	....SCC A	Windows prefetch file	PF
1A 00 00	...	Lotus Notes database template	NTF
1A 00 00 04 00 00		Lotus Notes database	NSF
1A 0x		LH archive file, old version (where x = 0x2, 0x3, 0x4, 0x8 or 0x9 for types 1-5, respectively)	ARC
1A 0B		Compressed archive file (often associated with Quake Engine games)	PAK
1A 35 01 00	.5..	GN Nettest WinPharaoh capture file	ETH
1F 8B 08	...	GZIP archive file	GZ
1F 9D 90		Compressed tape archive file	TAR.Z
21 12	!. .	AIN Compressed Archive	AIN
21 3C 61 72 63 68 3E 0A	!<arch>.	Unix archiver (ar) files and Microsoft Program Library Common Object File Format (COFF)	LIB
21 42 44 4E		Microsoft Outlook Personal Folder file	PST
23 20		Cerius2 file	MSI
23 20 4D 69 63 72 6F 73 6F 66 74 20 44 65 76 65 6C 6F 70 65 72 20 53 74 75 64 69 6F	# Micros oft Deve loper St udio	Microsoft Developer Studio project file	DSP
23 21 41 4D 52	# !AMR	Adaptive Multi-	AMR

		Rate (Algebraic Code Excited Linear Prediction) Codec, commonly audio format with GSM cell phones	ACELP
24 46 4C 32 40 28 23 29 20 53 50 53 53 20 44 41 54 41 20 46 49 4C 45	\$FL2@(#) SPSS DA TA FILE	SPSS Data file	SAV
25 21 50 53 2D 41 64 6F 62 65 2D 33 2E 30 20 45 50 53 46 2D 33 20 30	%!PS-Ado be-3.0 E PSF-3.0	Adobe encapsulated PostScript file (If this signature is not at the immediate beginning of the file, it will occur early in the file, commonly at byte offset 30)	EPS
25 50 44 46	%PDF <b>Trailers:</b> 0A 25 25 45 4F 46 0A (.%%EOF.) 0D 0A 25 25 45 4F 46 0D 0A (..%%EOF..) 0D 25 25 45 4F 46 0D (.%%EOF.)	Adobe Portable Document Format and Forms Document file	
28 54 68 69 73 20 66 69 6C 65 20 6D 75 73 74 20 62 65 20 63 6F 6E 76 65 72 74 65 64 20 77 69 74 68 20 42 69 6E 48 65 78 20		Macintosh BinHex 4 Compressed Archive	HQX
2A 2A 2A 20 20 49 6E 73 74 61 6C 6C 61 74 69 6F 6E 20 53 74 61 72 74 65	*** Ins tallatio n Starte d	Symantec Wise Installer log file	LOG

64 20			
[2 byte offset] 2D 6C 68	[2 byte offset] -lh	Compressed archive file	LHA, LZH
2E 52 45 43	.REC	RealPlayer video file (V11 and later)	IVR
1A 52 54 53 20 43 4F 4D 50 52 45 53 53 45 44 20 49 4D 41 47 45 20 56 31 2E 30 1A	.RTS COM PRESSED IMAGE V1 .0.	Runtime Software disk image	DAT
1D 7D	. }	WordStar Version 5.0/6.0 document	WS
2E 72 61 FD 00	.ra..	RealMedia streaming media file	RA
2E 73 6E 64	.snd	Sun Microsystems audio file format	AU
30	0	Microsoft security catalog file	CAT
30 00 00 00 4C 66 4C 65		Windows Event Viewer file	EVT
30 26 B2 75 8E 66 CF 11 A6 D9 00 AA 00 62 CE 6C	0&²u.ffi .ù.ª.bñl	Microsoft Windows Media Audio/Video File (Advanced Streaming Format)	ASF, WMA, WMV
30 31 4F 52 44 4E 41 4E 43 45 20 53 55 52 56 45 59 20 20 20 20 20 20 20	01ORDNA N CE SURVE Y	National Transfer Format Map File	NTF
31 BE or 32 BE	1¾ 2¾	Microsoft Write file	WRI
34 CD B2 A1	4Í² i	Extended tcpdump (libpcap) capture file (Linux/Unix)	
37 7A BC AF 27 1C	7z¾-'.	7-Zip compressed file	7Z
38 42 50 53	8BPS	Photoshop image file	PSD
3C	<	Advanced Stream redirector file. BizTalk XML-Data Reduced Schema file	ASX XDR
3C 21 64 6F 63 74 79 70	<!doctyp	AOL HTML mail file	DCI
3C 3F 78 6D 6C 20 76 65 72 73 69 6F 6E	<?xml ve rsion=	Windows Visual Stylesheet XML file	MANIFEST

3D			
3C 3F 78 6D 6C 20 76 65 72 73 69 6F 6E 3D 22 31 2E 30 22 3F 3E	<?xml ve rsion="1 .0"?>	XML User Interface Language file	XUL
3C 3F 78 6D 6C 20 76 65 72 73 69 6F 6E 3D 22 31 2E 30 22 3F 3E 0D 0A 3C 4D 4D 43 5F 43 6F 6E 73 6F 6C 65 46 69 6C 65 20 43 6F 6E 73 6F 6C 65 56 65 72 73 69 6F 6E 3D 22	<?xml ve rsion="1 .0"?>..< MMC_Cons oleFile ConsoleV ersion="	Microsoft Management Console Snap-in Control file	MSC
[24 byte offset] 3E 00 03 00 FE FF 09 00 06	[24 byte offset] >...þÿ..	Quattro Pro for Windows 7.0 Notebook file	WB3
3F 5F 03 00		Windows Help index file Windows Help file	GID HLP
[32 byte offset] 40 40 40 20 00 00 40 40 40 40	[32 byte offset] @0@ ..@0 @0	EndNote Library File	ENL
41 43 53 44	ACSD	Miscellaneous AOL parameter and information files	
41 4D 59 4F	AMYO	Harvard Graphics symbol graphic	SYW
41 4F 4C 44 42	AOLDB	AOL and AIM buddy list file	ABY, IDX
41 4F 4C 49 44 58	AOLIDX	AOL client preferences/settings file (MAIN.IND)	IND
41 4F 4C 49 4E 44 45 58	AOLINDE X	AOL address book index file	ABI
41 4F 4C 56 4D 31 30 30	AOLVM10 0	AOL personal file cabinet (PFC) file	n/a
41 72 43 01	ArC.	FreeArc compressed file	ARC
42 45 47 49 4E 3A 56 43	BEGIN:V C	vCard file	VCF

41 52 44 0D 0A 42 4C 49 32 32 33 51	ARD.. BLI223Q	Thomson Speedtouch series WLAN router firmware	BIN
42 4D	BM	Windows (or device-independent) bitmap image	BMP, DIB
42 5A 68	BZh	bzip2 compressed archive	BZ2, TAR.BZ2, TBZ2, TB2
43 42 46 49 4C 45	CBFILE	WordPerfect dictionary file (unconfirmed)	CBD
43 44 30 30 31	CD001	ISO-9660 CD Disc Image (This signature usually occurs at byte 8001, 8801, or 9001.)	ISO
43 4F 4D 2B	COM+	COM+ Catalog file	CLB
43 52 45 47	CREG	Windows 9x registry hive	DAT
43 52 55 53 48 20 76	CRUSH v	Crush compressed archive	CRU
43 57 53	CWS	Shockwave Flash file (v5+)	SWF
43 61 74 61 6C 6F 67 20 33 2E 30 30 00	Catalog 3.00.	WhereIsIt Catalog file	CTF
43 6C 69 65 6E 74 20 55 72 6C 43 61 63 68 65 20 4D 4D 46 20 56 65 72 20	Client U rlCache MMF Ver	IE History DAT file	DAT
44 42 46 48	DBFH	Palm Zire photo database	DB
44 4D 53 21	DMS!	Amiga DiskMasher compressed archive	DMS
44 4F 53	DOS	Amiga disk file	ADF
45 52 46 53 53 41 56 45 44 41 54 41 46 49 4C 45	ERFSSAVE DATAFILE	Kroll EasyRecovery Saved Recovery State file	DAT
45 56 46	EVF	EnCase evidence file	Enn (where nn are numbers)

46 41 58 43 4F 56 45 52 2D 56 45 52	FAXCOVER -VER	Microsoft Fax Cover Sheet		CPE
46 45 44 46	FEDF	(Unknown file type)	SBV	
46 4C 56	FLV	Flash video file	SWF	
46 4F 52 4D 00	FORM.	Audio Interchange File		AIFF
46 57 53	FWS	Shockwave Flash file		SWF
46 72 6F 6D 20 20 20 <b>or</b> 46 72 6F 6D 20 3F 3F 3F <b>or</b> 46 72 6F 6D 3A 20	FHom FHom ??? FHom:	A common file extension for e-mail files. Signatures shown here are for Netscape, Eudora, and a generic signature, respectively. EML is also used by Outlook Express and QuickMail		ELM
47 46 31 50 41 54 43 48	GF1PATCH	Advanced Gravis Ultrasound patch file		PAT
47 49 46 38 37 61 <b>or</b> 47 49 46 38 39 61	GIF87a GIF89a	Graphics interchange format file <b>Trailer:</b> 00 3B (.;)		GIF
47 50 41 54	GPAT	GIMP (GNU Image Manipulation Program) pattern file		PAT
47 58 32	GX2	Show Partner graphics file (not confirmed)		GX2
48 48 47 42 31	HHGB1	Harvard Graphics presentation file		SH3
49 20 49	I I	Tagged Image File Format file		TIF, TIFF
49 44 33	ID3	MPEG-1 Audio Layer 3 (MP3) audio file		MP3
49 49 2A 00	IT*.	Tagged Image File Format file (little endian, i.e., LSB first in the byte; Intel)		TIF, TIFF
49 53 63 28	ISc(	Install Shield v5.x or 6.x compressed file		CAB
49 54 53 46	ITSF	Microsoft HTML Help Compiled Help		CHM

Hex Signature	ASCII Signature	File Description	File Extension
49 6E 6E 6F 20 53 65 74 75 70 20 55 6E 69 6E 73 74 61 6C 6C 20 4C 6F 67 20 28 62 29	Inno Set up Unins tall Log (b)	Inno Setup Uninstall Log file	DAT
4A 41 52 43 53 00	JARCS.	JARCS compressed archive	JAR
4A 47 03 0E 00 00 00 or 4A 47 04 0E 00 00 00	JG..... JG.....	AOL ART file	ART
4C 00 00 00 01 14 02 00	L.....	Windows shortcut file	LNK
4C 01	L.	Microsoft Common Object File Format (COFF) relocatable object code file for an Intel 386 or later/compatible processors	OBJ
4C 4E 02 00	LN..	Windows Help file	HLP
4D 49 4C 45 53	MILES	Milestones v1.0 project management and scheduling software (Also see "MV2C" and "MV214" signatures)	MLS
4D 4D 00 2A	MM.*	Tagged Image File Format file (big endian, i.e., LSB last in the byte; Motorola)	TIF, TIFF
4D 4D 00 2B	MM.+	BigTIFF files; Tagged Image File Format files >4 GB	TIF, TIFF
4D 4D 4D 44 00 00	MMMD..	Yamaha Corp. Synthetic music Mobile Application Format (SMAF) for multimedia files that can be played on hand-held devices.	MMF
4D 53 43 46	MSCF	Microsoft cabinet file Powerpoint	CAB PPZ SNP

		Packaged Presentation Microsoft Access Snapshot Viewer file	
4D 53 46 54 02 00 01 00	MSFT....	OLE, SPSS, or Visual C++ type library file	TLB
4D 53 5F 56 4F 49 43 45	MS_VOICE	Sony Compressed Voice File Sony Memory Stick Compressed Voice file	CDR, DVF MSV
4D 54 68 64	MThd	Musical Instrument Digital Interface (MIDI) sound file	MID, MIDI
4D 56	MV	CD Stomper Pro label file	DSN
4D 56 32 43	MV2C	Milestones v2.1a project management and scheduling software (Also see "MILES" and "MV214" signatures)	MLS
4D 56 32 31 34	MV214	Milestones v2.1b project management and scheduling software (Also see "MILES" and "MV2C" signature)	MLS
4D 5A	MZ	Windows/DOS executable file.  MS audio compression manager driver.  Library cache file. Control panel application. Font file. ActiveX or OLE Custom Control. OLE object library. Screen saver.	COM, DLL, DRV, EXE, PIF, QTS, QTX, SYS  ACM  AX CPL  FON OCX  OLB SCR VBX

		VisualBASIC application. Windows virtual device drivers	VXD, 386
4D 5A 90 00 03 00 00 00	MZ.....	Acrobat plug-in DirectShow filter Audition graphic filter file (Adobe)	API AX FLT
4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF	MZ..... ....ÿÿ	ZoneAlam data file	ZAP
4D 69 63 72 6F 73 6F 66 74 20 56 69 73 75 61 6C 20 53 74 75 64 69 6F 20 53 6F 6C 75 74 69 6F 6E 20 46 69 6C 65	Microsof t Visual Studio Solution File	Visual Studio .NET Solution file	SLN
[84 byte offset] 4D 69 63 72 6F 73 6F 66 74 20 57 69 6E 64 6F 77 73 20 4D 65 64 69 61 20 50 6C 61 79 65 72 20 2D 2D 20	[84 byte offset] Microsof t Window s Media Player - -	Windows Media Player playlist	WPL
4E 41 56 54 52 41 46 46 49 43	NAVTRAFF IC	TomTom traffic data file	DAT
4E 45 53 4D 1A 01	NESM..	NES Sound file	NSF
4E 49 54 46 30	NITFO	National Imagery Transmission Format (NITF) file	NTF
4E 61 6D 65 3A 20	Name:	Agent newsreader character map file	COD
4F 50 4C 44 61 74 61 62 61 73 65 46 69 6C 65	OPLData baseFile	Psion Series 3 Database file	DBF
4F 67 67 53 00 02 00 00 00 00 00 00 00 00	OggS.....	Ogg Vorbis Codec compressed Multimedia file	OGA, OGG, OGV, OGX
4F 7B	O{	Visio/DisplayWrite 4 text file	DW4

		(unconfirmed)	
50 00 00 00 20 00 00 00	P.... . . .	Quicken QuickFinder Information File	IDX
50 35 0A	P5.	Portable Graymap Graphic	PGM
50 41 43 4B	PACK	Quake archive file	PAK
50 45 53 54	PEST	PestPatrol data/scan strings	DAT
50 49 43 54 00 08	PICT..	ADEX Corp. ChromaGraph Graphics Card Bitmap Graphic file	IMG
[92 byte offset] 51 45 4C 20	92 byte offset] QEL	Quicken data file	QEL
51 46 49 FB	QFI.	QEMU Qcow Disk Image	IMG
51 57 20 56 65 72 2E 20	QW Ver.	Quicken data file	ABD, QSD
52 45 47 45 44 49 54	RAZATDB1	Shareaza (Windows P2P client) thumbnail	DAT
52 45 47 45 44 49 54	REGEDIT	Windows NT Registry and Registry Undo files	REG, SUD
52 45 56 4E 55 4D 3A 2C	REVNUM:,	Antenna data file	ADF
52 49 46 46 xx xx xx xx 41 56 49 20 4C 49 53 54	RIFF.... AVI LIST	Windows Audio Video Interleave file	AVI
52 49 46 46 xx xx xx xx 51 4C 43 4D 66 6D 74 20	RIFF.... QLCMfmt	Compact Disc Digital Audio (CD- DA) file	QCP
52 49 46 46 xx xx xx xx 52 4D 49 44 64 61 74 61	RIFF.... RMIDdata	Windows Musical Instrument Digital Interface File	RMI
52 49 46 46 xx xx xx xx 57 41 56 45 66 6D 74 20	RIFF.... WAVEfmt	Audio for windows file	WAV
52 54 53 53	RTSS	Windows NT Netmon capture file	CAP
52 61 72 21 1A 07 00	Rar!...	WinRAR compressed archive file	RAR
53 43 48 6C	SCH1	Need for Speed:	AST

		Underground Audio file	
53 43 4D 49	SCMI	Img Software Set Bitmap	IMG
53 48 4F 57	SHOW	Harvard Graphics DOS Ver. 2/x Presentation file	SHW
53 49 45 54 52 4F 4E 49 43 53 20 58 52 44 20 53 43 41 4E	SIETRONI CS XRD S CAN	Sietronics CPI XRD document	CPI
53 49 54 21 00	SIT!.	StuffIt compressed archive	SIT
53 4D 41 52 54 44 52 57	SMARTDRW	SmartDraw Drawing file	SDR
53 51 4C 4F 43 4F 4E 56 48 44 00 00 31 2E 30 00	SQLOCONV HD...1.0.	DB2 conversion file	CNV
53 6D 62 6C	Smb1	(Unconfirmed file type. Likely type is Harvard Graphics Version 2.x graphic symbol or Windows SDK graphic symbol)	SYM
53 74 75 66 66 49 74 20 28 63 29 31 39 39 37 2D	StuffIt (c) 1997-	StuffIt compressed archive	SIT
55 43 45 58	UCEX	Unicode extensions	UCE
55 46 41 C6 D2 C1	UFAÆÓÁ	UFA compressed archive	UFA
55 46 4F 4F 72 62 69 74	UFOOrbit	UFO Capture v2 map file	DAT
56 43 50 43 48 30	VCPCH0	Visual C PreCompiled header file	PCH
56 45 52 53 49 4F 4E 20	VERSION	Visual Basic User-defined Control file	CTL
57 4D 4D 50	WMMP	Walkman MP3 container file	DAT
57 53 32 30 30 30	WS2000	WordStar for Windows Ver. 2 document	WS2
[29,152 byte offset] 57 69 6E 5A 69 70	[29,152 byte offset]	WinZip compressed archive	ZIP

		WinZip		
58 43 50 00	XCP	Cinco Network Sniffer, and Network Associates Sniffer capture file	NetXRay, General	CAP
58 50 43 4F 4D 0A 54 79 70 65 4C 69 62	XPCOM.Ty peLib	XPCOM type libraries for the XPIDL compiler		XPT
58 54	XT..	MS border	Publisher	BDR
5A 4F 4F 20	ZOO	ZOO compressed archive		ZOO
5B 47 65 6E 65 72 61 6C 5D 0D 0A 44 69 73 70 6C 61 79 20 4E 61 6D 65 3D 3C 44 69 73 70 6C 61 79 4E 61 6D 65	[General ]..Displ ay Name= <Display Name	MS Exchange 2007 extended configuartion file		ECF
5B 4D 53 56 43	[MSVC	Microsoft C++ Workbench Information File		VCW
5B 50 68 6F 6E 65 5D		Dial-up networking file (unconfirmed)		DUN
5B 56 45 52 5D 0D 0A 09 or 5B 76 65 72 5D 0D 0A 09 or	[VER] ... [ver] ...	AMU Pro document		SAM
5B 57 69 6E 64 6F 77 73 20 4C 61 74 69 6E 20	[Windows Latin	Microsoft Code Page Translation file		CPX
5B 66 6C 74 73 69 6D 2E 30 5D	[fltsim. 0]	Flight Simulator Aircraft Configuration file		CFG
5F 43 41 53 45 5F	_CASE_	EnCase case file (and backup)		CAS, CBK
60 EA	`ê	Compressed archive file		ARJ
63 75 73 68 00 00 00 02 00 00 00	cush.... ...	Photoshop Custom Shape		CSH
64 00 00 00	d...	Intel PROset/Wireless Profile		P10
64 73 77 66 69	dswfile	Microsoft Visual		DSW

6C 65 66 4C 61 43 00 00 00 22	fLaC..."	Studio workspace file Free Lossless Audio Codec file	FLAC
6C 33 33 6C	1331	Skype user data file (profile and contacts)	DBB
[4 byte offset] 6D 6F 6F 76  0x66-72-65-65 0x6D-64-61-74 0x77-69-64-65 0x70-6E-6F-74 0x73-6B-69-70	[4 byte offset] moov  Free mdat wide pnot skip	QuickTime movie file	MOV
72 65 67 66	regf	Windows registry hive file	DAT
72 74 73 70 3A 2F 2F	rtsp://	RealMedia metafile	RAM
73 6C 68 21 or 73 6C 68 2E	slh! slh.	Allegro Generic Packfile Data file (0x21 = compressed, 0x2E = uncompressed)	DAT
73 72 63 64 6F 63 69 64 3A	srcdocid :	CALS raster bitmap file	CAL
73 7A 65 7A	szez	PowerBASIC Debugger Symbols file	PDB
75 73 74 61 72	ustar	Tape Archive file	TAR
76 32 30 30 33 2E 31 30 0D 0A 30 0D 0A	v2003.10 .0...	Qimage filter	FLT
78	x	Mac OS X Disk Copy Disk Image file	DMG
7B 0D 0A 6F 20	{ ..o	Windows application log	LGC, LGD
7B 5C 72 74 66 31	{\rtf1 Trailer: 5C 70 61 72 20 7D 7D (\par } })	Rich text format word processing file	RTF
7E 42 4B 00	~BK.	Corel Paint Shop Pro image file	PSP
7F 45 4C 46	.ELF	Executable and linking format in Linux/unix	n/a
80		Relocatable object	OBJ

		code	
80 00 00 20 03 12 04		Dreamcast audio file	ADX
89 50 4E 47 0D 0A 1A 0A		Portable Network Graphic File	PNG
8A 01 09 00 00 00 E1 08 00 00 99 19	.....á. ....	MS Answer Wizard file	AW
91 33 48 46	'3HF	Hamarsoft HAP 3.x compressed archive	HAP
95 00 or 95 01	.. ..	PGP secret keying file	PKR
9C CB CB 8D 13 75 D2 11 91 58 00 C0 4F 79 56 A4	.EE..UÒ. .X.ÀOyV¤	Outlook address file	WAB
[512 byte offset] A0 46 1D F0	[512 byte offset] F.Ø	PowerPoint presentation subheader (MS Office)	PPT
A1 B2 C3 D4	i²ÀÓ	tcpdump (libpcap) capture file (Linux/Unix)	
A1 B2 CD 34	i²Í4	Extended tcpdump (libpcap) capture file (Linux/Unix)	
A9 0D 00 00 00 00 00 00	©.....	Access Data FTK evidence file	DAT
AC 9E BD 8F 00 00	¬.¾... ...	Quicken data file	QDF
B1 68 DE 3A	±hP:	Graphics Multipage PCX bitmap file	DCX
B5 A2 B0 B3 B3 B0 A5 B5	µ¢°³³°¥µ	(Unknown file type...)	CAL
BE 00 00 00 AB 00 00 00 00 00 00 00 00	¾...«... ....	MS Write file	WRI
C3 AB CD AB	Ã«Í«	MS Agent Character file	ACS
C5 D0 D3 C6	ÃÐÓÆ	Adobe encapsulated PostScript file	EPS
CA FE BA BE	Êþ°¾	Java bytecode file	CLASS
CD 20 AA AA 02 00 00 00	Í a a ...	Norton Anti-Virus quarantined virus file	n/a
CF 11 E0 A1 B1 1A E1 00	Í.à;±.á.	Perfect Office document [Note similarity to MS Office header, below]	DOC

CF AD 12 FE	í.p	Outlook Express e-mail folder	DBX
D2 0A 00 00	Ø...	GN Nettest WinPharaoh filter file	FTR
D4 2A	Ô*	AOL history (ARL) and typed URL (AUT) files	ARL, AUT
D7 CD C6 9A	×ÍÆ.	Windows graphics metafile	WMF
DC DC	ÜÜ	Corel color palette file	CPL
DC FE	Üþ	eFax file format	EFX
E3 82 85 96	â...	Windows password file	PWL
EB 3C 90 2A	ë<.*	GEM Raster file	IMG
[512 offset] EC A5 C1 00	[512 byte offset] ìÝÅ.	Word document subheader (MS Office)	DOC
ED AB EE DB	í"îÛ	RedHat Package Manager file	RPM
[512 offset] FD FF FF FF nn 02		Excel spreadsheet subheader (MS Office) (where nn = 0x10, 0x22, 0x23, 0x28, or 0x29)	XLS
[512 offset] FD FF FF FF 20 00 00 00	[512 byte offset] Ýÿÿÿ ...	Developer Studio File Workspace Options subheader (MS Office).  Excel spreadsheet subheader (MS Office)	OPT  XLS
512 offset] FD FF FF FF xx xx xx xx xx xx xx xx 04 00 00 00	[512 byte offset] Ýÿÿÿ..... .....	Thumbs.db subheader (MS Office)	DB
FF	ÿ	Windows executable (SYS) file	SYS
FF 00 02 00 04 04 05 54 02 00	ÿ..... .T ..	Works for Windows spreadsheet file	WKS
FF 46 4F 4E 54	ÿFONT	Windows international code page	CPI

FF 4B 45 59 42 20 20 20	ÿKEYB	Keyboard driver file	SYS
FF 57 50 43	ÿWPC	WordPerfect text and graphics file	
FF Ex FF Fx	ÿ. ÿ.	MPEG audio file frame	MPEG, MPG, MP3
FF FF FF FF	ÿÿÿÿ	DOS system driver	SYS
FF FE 23 00 6C 00 69 00 6E 00 65 00 20 00 31 00	ÿþ#.l.i. n.e..1.	Windows MSinfo file	MOF

# Index

- ADD 9
- AES 58
- Assembly 1
- ASLR 51
- Application Programming Interface 47
- Asymmetric Encryption 67
- Algorithm Provider 63
- Arithmetic Operation in Assembly 21
- Auxiliary Segment Register 2
- Array in Assembly 24
- Anti Debugging 140
- Anti Disassembly 128
- Anti – Reversing technique 127
- AVI 99
- Base Relocation Table 73
- BCrypt 59
- Bound Import Table 74
- Blind Return 124
- Break Points 144
- Call Register 125
- Calling Convention 16
- Calculating Offset 126
- Carry Flag 2
- Cdecl calling convention 17
- Code Segment Register 1
- Code Emulation 162
- Context Switching 43
- CmLogLevel 50
- CmLogLevelSet 50
- CNG 58
- Critical section 45
- Cryptographic Agility 59
- Data Segment Register 1
- Data Constructs 17
- Dos Header 70
- DES 58
- Debugging Information 73
- Detecting hardware Breakpoint 144
- Division 23
- Direct Jump 126
- Dlmalloc 96
- ELF Header 79
- Executable Format 79
- Executable Data Section 20
- ExpEchoPoolCalls 49
- Exploiting SEH 119
- Evasion of Disassembler 128
- Events 45
- Flag Register 2
- Fast call calling convention 17
- Format String 112
- Floating point instructions 11
- Global Descriptor Table 145
- Global Variables 18
- GS 55
- Hardware breakpoint 143
- Heap Overflow 94
- Hash Functions 64
- Heap Defenses 53
- Imported Variables 19
- Import Redirection 162
- Import Table 73
- Import Table Reconstruction 158
- Integer Overflows 106
- Interrupt Flag 3
- Interrupt Descriptor Table 145
- Injection Techniques 123
- IO Flag 3
- Jmp 7
- Kernel Memory Management 37

- Linking File Format 79  
Linked List 26  
Linear Sweep Disassembler 130  
Local Variable 19  
Lock and Repeat Prefix 4  
LpcpTraceMessage 50
- Memory Management 29  
Metered Section 46  
MmDebug 49  
Modulo 24  
Mode Switches 43  
Multiplication 22  
Mutexes 45
- Named Object 41  
NtGlobalFlag 49  
NOP 9  
No Operation Sled 125  
Non executable Memory 94  
NX 54
- Objects and Handles 40  
ObpShowAllocAndFree 50  
Off-by-One Overflow 90  
Opcode  
Original First Thunk 158  
Overflow Flag 3
- Paged Memory Management 36  
Parity Flag 2  
Pointer Encoding 57  
Pop Return 126  
PE file format 69  
Processes 41  
Process Initialization Sequence 46
- Random Number Generator 64  
RSA 58  
Recursive Traversal Disassembler 132  
Register 1,19  
Representation of class in assembly 27  
Reversing Windows NT 48
- Section Object 39  
Security in Vista 50  
SepDumpSD 50  
Semaphore 46  
Segment Override Prefixes 5  
Segmented Memory Management 34  
Self-modifying code 137  
Sign Flag 2  
Structure Exception Handler 116  
Signature and Verification 68  
Software Breakpoint 142  
Stack Segment Register 2  
Stdcall calling convention 17  
Stack Checking 90  
Stack Overflow 85  
Stack Setup 13  
Stack Randomization 52  
Sub 9  
Symmetric Encryption 65  
Synchronization Objects 44
- Thiscall calling convention 18  
Threads 41  
Thread Storage Table 73  
Threaded Local Storage 20
- Unpacking 147  
User Memory Management 37  
User Mode Address Space 39
- Virtual Address Descriptor 39  
Vista 50  
Virtual Function 26  
Virtual Machine De 6  
Virtual Memory Management 29  
Virtual Machine Obfuscation 139
- Windows NT 49
- Zero Flag 2
- 80x86 Instruction Format 3