Windows Malware Analysis Essentials

This book will start with the basics of computing fundamentals such as number systems and Boolean algebra. Further, you'll learn about x86 assembly programming and its integration with high-level languages such as C++. You will understand how to decipher disassembly code obtained from the compiled source code and map it back to its original design goals. By delving into end-to-end analysis with real-world malware samples to solidify your understanding, you'll sharpen your technique of handling destructive malware binaries and vector mechanisms. You will also be encouraged to consider analysis lab safety measures so that there is no infection in the process.

Finally, we’ll have a well-rounded tour of various emulations, sandboxing, and debugging options so that you know what is at your disposal when you need a specific kind of weapon in order to nullify the malware.

Who this book is written for

This book is best for someone who has prior experience with reverse engineering Windows executables and wants to specialize in malware analysis. The book presents the malware analysis thought process using a show-and-tell approach, and the examples included will give any analyst confidence in how to approach this task on their own the next time around.

What you will learn from this book

- Use the positional number system for clear conception of Boolean algebra that applies to malware research purposes
- Get introduced to static and dynamic analysis methodologies and build your own malware lab
- Analyze destructive malware samples from the real world (ITW) from fingerprinting and static/dynamic analysis to the final debrief
- Understand different modes of linking and how to compile your own libraries from assembly code and integrate the code in your final program
- Get to know about the various emulators, debuggers and their features, and sandboxes and set them up effectively depending on the required scenario
- Deal with other malware vectors such as PDF and MS Office-based malware as well as scripts and shellcode

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Down the Rabbit Hole'
- A synopsis of the book’s content
- More information on Windows Malware Analysis Essentials
Victor Marak is a security researcher, an electronic musician, and a world backpacker. He is a college dropout and an autodidact, and he loves working on interesting subjects such as medieval music composition, demonology, DSP electronics, and psychology. He has worked for start-ups, mid-tier, and fortune 500 companies with 5 years of experience in anti-virus technologies and malware research. He was into music production prior to joining the anti-malware industry, and his solo projects are on the world's largest electronic dance music market—Beatport, as well as other major retailers like iTunes, Amazon and Traxxsourse. He is in perpetual backpacking mode, set to globe-trotting, especially to his favorite countries in Europe and Russia. He can be found hanging around in the wrong social networks - LinkedIn and Quora.

This is his first book.
Welcome to *Windows Malware Analysis Essentials*. This book will help you demystify the process of analyzing Windows-specific malware, and it will show you how to work with the weapons in the malware analysts' arsenal. It will also help you develop skills to analyze malware on your own with informed confidence.

Malware is a big and global business—with malware fighters a relatively reclusive and closed community since the inception of the antivirus industry. This also means that anti-malware technologies are a relative mystery to most regular folk with a dichotomy existing perpetually. Only recently have extensive steps been taken to alleviate this problem, which is becoming more and more visible and pervasive. Even gaining knowledge has become an expensive affair with training and courses running into many thousands of dollars for relatively foundational information. The training market does have value and an audience, but the IT masses do not have much access to it, even if the interest is there. Malware has moved on from being a sport or hobby to organized crime and even though the hacker community shares between them, the IT crowd is not very initiated or well informed. Skilled manpower is required, and right now, demand exceeds supply. Working in an anti-malware firm is not the only way to fight malware, and with signature-based detection slowly becoming an unwieldy technology, more minds are required to innovate or invent new solutions to existing challenges. This has to be a multipronged approach taking from data analytics, mathematics, biology, law enforcement, and of course, computers, among a host of other requirements. Getting up to speed with the fundamentals of malware analysis makes things more manageable when the proverbial stuff hits the fan.
The book will commence with the essentials of computing where you get a foothold for the challenges ahead. It will show you how to decipher disassembly text obtained from analysis of compiled binary code and acclimatize you to the battery of tools at your disposal. It will also give you an unprecedented look at the myriad ways that an analyst can approach analyses of real-world malware and points you in the right direction in order to start building your own malware lab, gathering intelligence, and revealing malevolent agents through thorough investigation. This book will, as a rite of passage, effectively prepare you to be the anti-malware warrior you always wanted to be.

What this book covers

Chapter 1, *Down the Rabbit Hole*, prepares you for the challenges ahead by reviewing some essential computing concepts, which must be mastered before you commence analysis of malware. You will explore number bases, binary arithmetic, and boolean algebra. This chapter also covers the malware analysts' toolkit and introduces IDA Pro, the Portable Executable format, and instances of reverse engineering program binaries on the Windows platform. This will set the pace for the activities in the chapters ahead.

Chapter 2, *Dancing with the Dead*, covers x86 assembly programming using VC++ 2008 and MASM32. You will then proceed with x86 disassembly of compiled code binary and analysis thereof in VC++ IDE. Finally, you will explore the myriad configurations in order to do assembly programming in the VC++ environment and end with a detailed overview of common data structures and code constructs in the C and x86 assembly.

Chapter 3, *Performing a Séance Session*, demonstrates a complete end-to-end malware analysis of real-world destructive malware. You will get unprecedented insight into an analysis session along with configurations, tips and tricks, and step-by-step progression towards a full analysis, right up to signature generation and report creation for the entire set of malware samples.

Chapter 4, *Traversing Across Parallel Dimensions*, delves into kernel-mode concepts and the fundamentals of Windows internals, which will help you with your analysis and understanding of the overall framework you are dealing with. You will work with IDA Pro and Windbg as the primary weapons for kernel mode analysis.
Chapter 5, *Good versus Evil – Ogre Wars*, rounds off the earlier excursions with a general set of devices—from the configuration of the Linux virtual machine guest for wiretapping the network activity of malware, to exploring XOR deobfuscations programmatically. Thereafter, you will revisit malware analysis with a different target—malicious web scripts, and you will learn how the innards are picked one by one, gathering information about the exploits used, the various infection vectors, dealing with obfuscated JavaScript and working with a rather familiar set of new tools. You will also be introduced to Mandiant Redline for malware forensics, and finally end the tour with a discussion of bytecode decompilation utilities and open source tools for malware intelligence gathering.
Before we get started with analyzing malware, you need to start at the baseline, which will involve reviewing some fundamental tenets of computer science. Malware analysis essentially deals with an in-depth investigation of a malicious software program, usually in some binary form procured through collection channels/repositories/infected systems or even your own Frankenstein creations in a lab. In this book, we focus on Windows OS malware and the myriad methods and the inventory required for their analyses. Much like a time and space tradeoff for computer algorithms (and the infinite monkeys with typewriters paradigm), the analyst must be aware that given enough time, any sample can be analyzed thoroughly, but due to practical constraints, they must be selective in their approach so that they can leverage the existing solutions to the fullest without compromising on the required details. If churning out anti-virus signatures for immediate dispersal to client systems is the priority, then finding the most distinguishing characteristic or feature in the sample is a top priority. If network forensics is the order of the day, then in-depth packet traces and packet analyses must be carried out. If it’s a memory-resident malware, then malware memory forensics has to be dealt with. Likewise, in unpacking an armored sample, fixing the imports/exports table to get a running executable might not be the best use of your time, as if the imports are functional in memory and the details are available, investigation of the Modus Operandi (MO) must be the primary focus and not memory carving, particularly if time is a factor. Perfectionism in any process has its benefits and liabilities. Malware analysis is both a science and an art. I believe it is more like a craft wherein the tools get the work done if you know how to use them creatively, like a sculptor who has a set of mundane chisels to remove stone chips and etch a figure of fantasy out of it. As any artist worth his salt would say, he is still learning his craft.
The primary topics of interest for this primer are as follows:

- Number systems
- Base conversion
- Signed numbers and complements
- Boolean logic and bit masks
- Malware analysis tools
- Entropy

The motivation behind these topics is simple: if these fundamentals are not clear, reading hex dumps and deciphering assembly code will be a pain in the neck. It is vital that you know these topics like the back of your hand. More importantly, I believe that understanding the concepts behind them may help you understand computers as a whole more intimately in order to deal with more complex problems later on. There is no silver bullet for malware analysis methodologies as quite a lot of problems that surface are related to computing boundaries and are NP-complete, much like an irreversible chemical process or an intractable problem. You will be using debuggers, disassemblers, monitoring software, visualization, data science, machine learning, regular expressions (automata), automation, virtualization, system administration, the software development tool chain and system APIs, and so on.

Thus, you have a set of tools that enable you to peek into the coexisting layers and a set of techniques that enable you to use these tools to an optimum level. Also, you have to wear many hats—things like forensics, penetration testing, reverse engineering, and exploit research blur the line when it comes to malware technologies that are in vogue, and you have to keep up. The rest comes with experience and tons of practice (10,000 hours to mastery according to Outliers by Malcolm Gladwell).

There is no shortcut to hard work, and shortcuts can be dangerous, which ironically is learned from experience many times. The primer will be quick, and it will be assumed that you have a solid understanding of the topics discussed before you read the following chapters, particularly x86/x64 assembly and disassembly. From here, you will proceed to x86/x64 assembly programming and analysis, static and dynamic malware analysis, virtualization, and analysis of various malware vectors.

**Number systems**

The number system is a notational scheme that deals with using a symbol to represent a quantity.
Chapter 1

A point to ponder: We know that a quantity can be both finite and infinite. In the real world, many things around us are quantifiable and can be accounted for. Trees in a garden are finite. The population of a country is finite. In contrast, sand particles are seemingly infinite (by being intractable and ubiquitous). Star systems are seemingly infinite (by observation). Prime number sequences are infinite (by conjecture). It is also understood that tangible and intangible things exist in nature in both finite and infinite states. A finite entity can be made infinite just by endless replication. An infinite and intangible entity can be harnessed as a finite unit by giving it a name and identity. Can you think of some examples in this universe (for example, is this one of many universes or is it the only one and infinitely expanding)?

In my experience, there is a lot of confusion regarding number systems, even with some experienced IT folk. Quantities and the representation of these quantities such as symbols/notations are primarily separate entities. A notation system and what it represents are completely different things, although because of ubiquity and visibility, the meanings are exchanged and we take it for granted that they are both one and the same, and that creates the confusion. We normally count using our fingers because it seems natural to us. We have five digits per hand and they can be utilized to count up to 10 units. So, we developed a decimal counting system. Note that the numbers 0 to 9 constitute the whole symbol set for whole numbers. While defining a symbol set, although we use the symbols that are designed through the centuries that have passed and have their place, it is not mandatory to define numbers only in that particular set. Nothing prevents us from developing our own symbol set to notate quantities.

An example symbol set = [NULL, ALPHA, TWIN, TRIPOD, TABLE], and we can substitute pictures in the above set, which directly map to {0, 1, 2, 3, 4}. Can you think of your own symbol set?

The largest number in the symbol set (9) is one less than that of the base (10). Also, zero was a relatively late addition, with many cultures not realizing that null or nothing can also be symbolized. Using zero in a number system is the crux to developing a position-based encoding scheme. You can only occupy something where there is a void that acts as a container, so to speak. So, you can think of 0 as a container for symbols and as a placeholder for new positions. In order to count 10 objects, we reuse the first two symbols from the symbol set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. This is key to understanding number systems in vogue. What happens is that each number position/column is taken as an accumulator. You start at your right and move towards the left. The value in the first column changes in place as long as the symbol set has a symbol for it.
When the maximum value is achieved, a new column toward the left is taken and the counting continues with the initial symbols in the symbol set. Thus, think of each column as a bucket that is a larger bucket than the one to the right of it. Further, each new column represents the largest quantity of the last column. Here, the number of columns used or the number of symbol places denotes the range of quantities that can be represented. We can only use the symbols in the symbol set. Thus, if we had a set of infinite symbols for each quantity, we would not have to reuse the symbols to represent larger quantities, but that would be very unwieldy as we humans don’t have a very good memory span.

To reiterate, think of the columns as containers. Once you are out of symbols for that particular column, you reuse the first symbol greater than zero. Thereafter, you reset the previous column to zero, and start increasing the symbol count till it reaches the maximum in the set. You then repeat the process for the specific quantity to be represented. Study the following image to gain more understanding visually:
You can also look at the number system notation as a histogram-based notation that uses symbols instead of rectangles, wherein the quantity is represented by the total count in a compact representation. The histogram is essentially a statistical tool to find the percentage of an entity or a group of entities and other control points such as features of the entities in a population that contains the entities.

Think of it as a frequency count of a particular entity. Here, each entity refers to the base power group that each digit towards the left represents.

So, taken as a summation of weights, each position that can be seen as representing a total Frequency Count of how many of that position's relative quantity. Instead of drawing 15 lines to denote 15 objects, we use the symbols 1 and 5 to denote 5 objects and 10 more, with 5 joining the 1 and then taking the place of 0, which acts as a container or placeholder to give the combined notation of 15.

For a larger number such as 476, you can see this notation as a count of how many 100s, 10s, and the core symbol set values. So, \(400 = 4 \times 100\) or there are 4 hundreds, and \(7 \times 10\) or that there are 7 tens and 6 values more. The reason you add these up is because they each represent a part of the total.

Can you repeat the process with a new base? Why don't you try base 3? The solution will be given later in this chapter, but you must try it yourself first.

Have you ever wondered why the base 8 (octal) does not have the numbers 8 and above in its notation? Use the same rules that you have just read to reason why this notation system works the way it does. It follows the same number symbol-based position-relative notation. You can also think of this as weights being attached to the positions as they are positioned towards the left. Finally, as each row signifies a count of the next quantity, you essentially sum up all the position values according to their weights.

We are accustomed to using the above formula as an automated response for counting numbers without ever giving much thought to the reasoning behind this notational system. It's taken for granted that you never question it.

The hexadecimal base notation also works in the same way. The reasoning behind using 16 as a quantity belies the fact that a permutation of 2 symbols \(\{0, 1\}\) to a length of 4 gives us 16 different patterns. Since 4 bits used as a basic block works for grouping bit sequences as per engineering conventions, the nibble is the smallest block unit in computing. The bit is the smallest individual unit present. The minimum value is 0000 and the largest value is 1111. These unique patterns are represented using symbols from the alphabet and the numbers 0 to 9.
You can replace the alphabet symbols A to F with any shape, picture, pattern, Greek letter, or visual glyph. It’s just that the alphabets are already a part of our communication framework, so it makes sense to reuse them. So, the convention of grouping 4 bits to form a pattern using a notation that expresses the same thing provides a much more convenient way to look at binary information. Since our visual acuity is much sharper when we form groups and patterns, this system works for engineers and analysts, who need to work with binary data (in a domain-agnostic manner) on a regular basis. Note that as per convention, hexadecimal values are prefixed with 0x or post-fixed with H to denote hexadecimal notation.

The hexadecimal symbol set = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

| List of 16 values from 0 and 1 symbols: |
| Symbol set {0,1}, notation length = 4 |
| 0000 = 0x0 |
| 0001 = 0x1 |
| 0010 = 0x2 |
| 0011 = 0x3 |
| 0100 = 0x4 |
| 0101 = 0x5 |
| 0110 = 0x6 |
| 0111 = 0x7 |
| 1000 = 0x8 |
| 1001 = 0x9 |
| 1010 = 0xA |
| 1011 = 0xB |
| 1100 = 0xC |
| 1101 = 0xD |
| 1110 = 0xE |
| 1111 = 0xF |
Permutations are also the foundational mathematics behind data type representation. So, we have taken 4 bits to form a nibble and 8 bits to form a byte. What is a byte? Taken simply, it is a series of symbols from the set \{0, 1\} to a length of 8, which represents the permutated value of that particular sequence as a quantity. It could also be further used to represent a packed data type where each bit position denotes a toggle of a value as on or off, similar to an array of switches. Since we work with programming languages, the binary data types are of a primary interest as they work like an index into a table where the entire range from the minimum to the maximum is already worked out as part of a finite series. Thus, the bit pattern 0001111 gives the value of 15 out of a total of 2^8 values. Why 2^8? This is because when you need to compute unique values out of a symbol set to a specific length, you take the total number of symbols to the power of the length to get the maximum value. However, you also have to take into account the primary conditions for permutations and its difference from combinations all relating to symbol usage and being ordered or not. As a rule, to reuse the symbols in a specific order, you can take powers, as in the case of permutations. However, if using a symbol removes it from the participation of the next position's symbol set, you need to take factorials, as in the case of combinations. They all fall into a branch of mathematics called Combinatorics. Likewise, do you see the logic behind primitive data types such as int, short, char, and float? When using custom data types, such as structs and classes, you are effectively setting up a binary data structure to denote a specific data type that could be a combination of primitive data types or user-defined ones. Since the symbol set is the same for both primitive/primary and data types, it is the length of the data structure assigned per data type that gives meaning to the structure.

For a simple exercise, find the unique ways in which you can arrange the letters \{A, B, C\}, where each symbol can be reused to a length of 3, that is, each position can use any symbol from the set above. Thereafter, find the unique ways in which you can combine the symbols, without repeating any previous pattern but in any sequence. You will find that you get 27 patterns from the first exercise and 6 patterns from the second. Now, build a formula or try to model this pattern. You get \((base^{(pattern length)})\) and \(factorial\ (base)\). This is how binary notation is used to encode quantities, which are being denoted by symbols (which can also be mapped to a scheme), which in turn are based on the principles of human language, and therefore, all information can be encoded in this manner.
Computers do not even understand the symbol 1 (ASCII 0x31) and the symbol 0 (ASCII 0x30). They only work with voltage levels and logic gates as well as combined and sequential circuits such as D flip-flops for memory. This complex dance is orchestrated by a clock that sets things in motion (a regular pulse of \(n\) cycles/s aids in encoding schemes, in much the same way as in music, the rhythm brings predictability and stability that greatly simplifies encoding/decoding and transmission); much like a conductor, in tandem with the microprocessor, which provides built-in instructions that can be used to implement the algorithm given to it. The primary purpose of using various notation systems is that doing so makes it more manageable for us to work with circuit-based logic and provides an interface that looks familiar so that humans can communicate with the machine. It’s all just various layers of abstraction.

The following table shows how the base 3 notation scheme can be worked out:

<table>
<thead>
<tr>
<th>Base 3 Notation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol set = ({0, 1, 2})</td>
</tr>
<tr>
<td>0=0</td>
</tr>
<tr>
<td>1=1</td>
</tr>
<tr>
<td>2=2</td>
</tr>
<tr>
<td>3=10</td>
</tr>
<tr>
<td>4=11</td>
</tr>
<tr>
<td>5=12</td>
</tr>
<tr>
<td>6=20</td>
</tr>
<tr>
<td>7=21</td>
</tr>
<tr>
<td>8=22</td>
</tr>
<tr>
<td>9=100</td>
</tr>
<tr>
<td>10=101</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>As a system of positional weights:</td>
</tr>
<tr>
<td>(1 \times 3^2 + 0 \times 3^1 + 1 \times 3^0 = 10)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>As a histogram sigma of each column:</td>
</tr>
<tr>
<td>1st column symbol=0. That means the number of singular units till the maximum number of symbols in the set has been crossed so the count at this point is at least 3. Note that 0, 1 and 2 are the symbols exhausted for this count. If the number of units were 2, this column would contain 2 and there would be no 2nd column.</td>
</tr>
<tr>
<td>2nd column to the left symbol=1. That means that the number of units have exceeded the symbol set count till 2 units and now, the zero placeholder is used to denote the 3rd unit in count, with the zero in the 1st column signifying that it is ready to accumulate increments till the 2nd column is also exhausted. After this both will reset to zero and the 3rd column gets active.</td>
</tr>
<tr>
<td>So looking at the symbol 10, its a count of the column value and how much it represents. The count therefore is 3.</td>
</tr>
</tbody>
</table>
Can you write a program to display the base notation of bases from 2 to 16? A maximum of base 36 is possible by using alphabets and numbers as the symbol set after which new symbols or combinations of existing symbols have to be used to map a new symbol set. I think that this would be a great exercise in programming fundamentals.

**Base conversion**

You have seen how the positional notation system is used to represent quantities. How do you work with the myriad bases that are developed from this scheme? Converting decimal to binary and binary to hexadecimal or vice versa in any combination must be a workable task in order to successfully make use of the logic framework and communicate with the system.

**Binary to hexadecimal (and vice versa)**

This is the simplest base conversion method once you get the hang of it. Each hexadecimal digit maps directly to a specific binary pattern. Dividing any binary pattern into multiples of 4 gives us the corresponding hexadecimal form. If less than 4 bits are used, 0 is left padded (for instance, 11 0011 0101 gets left padded to 0011 0011 0101 in order to get 3 nibbles) to get it to 4 bits or a multiple length thereof. Likewise, for larger lengths but ending at odd positions, zero is padded again to get the length of a multiple of 4. Remember that each character in the hexadecimal representation is a nibble. Hence, larger composite data types are grouped according to the data type length. WORD has 2 bytes, and DWORD has 4 bytes. These terms relate to data types or for our purposes, the number of bits used to collectively represent a unit of data—exhibiting properties of the total pattern quantity count and the placeholders for each of the individual patterns. These directly map to a value in the data type range; for instance, a pattern length of 16 bits is conventionally called a WORD, which gives a total pattern value count of $2^{16}$ values, and the value 2, for instance, can be represented in 16 bits as 0000 0000 0000 0010, which directly maps to the value 2 from a range of 0 to 65,535. The processor WORD is normally the most fundamental data unit that is used in the processor architecture. In IA-32, the **natural or processor word** is taken as 32-bit units and other data types derived from it. It can also conventionally mean the type of an integer implemented in the architecture. Refer to [https://en.wikipedia.org/wiki/Word_(computer_architecture)] for a more general overview. Similarly, for any hexadecimal number, just map each of its characters to the 16 different binary values and concatenate them in order to get the resulting binary sequence.

1111 1101 <-> FDh [byte data type]
Decimal to binary (and vice versa)

Binary to decimal is achieved by adding the weights for each bit position that is set and adding them up.

Decimal to binary requires you to divide the number by 2 and set the symbol for any remainder and 0 for no remainder after every step, and recursively do the division till you get to 2 or below as the dividend. Essentially, you take stock of the modulus of the entire process in a stack data structure and concatenate them in reverse to get the resulting binary value.

For instance, to convert 9 decimal to binary, notice the modulus or remainder:

- 9/2 = 4 with remainder 1
- 4/2 = 2 with remainder 0
- 2/2 = 1 with remainder 0
- ½ = 0 with remainder 1

Reading in reverse, that is, bottom to top, we get 1001, which if you multiply the places in powers of 8 would yield $1 \times 2^3 + 0 + 0 + 1 \times 2^1 = 8 + 1 = 9$. Mapping 1001 to hexadecimals will still give you 0x9 as after that, the symbol set for quantities above 9 is letters.

The divisions by base till you reach the base value and record the modulus method as well as the add the integer powers of the base to get the result method are the most prevalent in computing and work with every base that subscribes to this positional notational system.

Try doing converting decimal values to hexadecimals (Hint: Divide by 16 and take the modulus/add the powers of each hexadecimal character decimal value (nibble representation) and multiply with each power of 16.).

Octal base conversion

Octal is a legacy form and is not used much nowadays in our current technological setup. However, now, you know how to deal with it. The simple way to break a binary pattern into its octal representation is to group the bits into groups of three and write the decimal number for that pattern. Why 3 you ask? It is octal, so 8 is the base of the notation. Taking a binary of length 3 and setting each bit position to 1 each to get 111 gives us 7 in decimal. This is the maximum value that will be represented by the symbol set (remember how the positional/placeholder-based notation works). Thus, number symbol patterns of a length of 3 places are enough to realize the entire symbol set of the octal base. Hence, you start by grouping bits into groups of length 3.
Chapter 1

Signed numbers and complements

An annoying topic for many is negative numbers. Their representation in binary is a set of workaround techniques to represent negative numbers with the same data types and symbol set. How would you differentiate the values in that scenario? A binary pattern is by itself quite neutral to begin with. It is a representation of a sequence of symbols that have two possible values from the symbol set, which have a final resulting value based on a particular permutation pattern that denotes this value. In essence, the binary pattern could be a number, a picture, a text file, a video file, or so on. What a pattern constitutes is also dependent on who looks at it and how. Inherently, the pattern is quite ambiguous without a context to give it its definite meaning. Hence, in terms of compiled machine code, which we will dealing with, the way the instructions and their opcodes are chosen by the compiler build a context around the regular data type, for instance, a DWORD, which is 32 bits long, or a WORD, which is 16 bits long. This sort of structure prevents ambiguity for the translation mechanisms in place. You will learn ahead in assembly programming that the compiler will choose certain instructions based on its inferred data type. Thus, context is supported by design. JAE and JGE is some examples using analogous instructions, where the value for the first instruction mnemonic denotes the use of unsigned numbers, whereas the second instruction mnemonic denotes the use of signed numbers.

![Diagram of bit weights and MSB/LSB](image-url)
Signed data types will effectively halve the range of the unsigned data type version. This is because of the use of the sign bit as the **Most Significant Bit (MSB)**. The binary values that will be represented will use 7 bits, which is $2^7$ for a signed byte and $2^31$ for a signed DWORD. The largest positive value will be $(2^{n-1}-1)$. So, for a byte, the largest positive value will be $2^7 - 1 = 127$. However, the largest negative value will be -128. In binary patterns, since each position is a power of 2, using one less bit toward the left (direction of increment) will result in using half of the value (shift left is multiplication by 2 and shift right is division by 2). Now, anytime, you see the familiar formula of $(2^n - 1)$, you know that it is essentially the maximum value of that data type (n bits), which is the last value in the permutation chain. $2^n$ will represent the total number of values that you can use including zero. You will see this formula in lots of computing areas, including the area of finding large prime numbers, which is an active area of research.

The main methods used are sign and magnitude, where the MSB is set to denote negative numbers and 1's complement and 2's complement where the complement is taken by inverting the value (1's complement, NOT x86 instruction) and adding 1 to the result to get the 2's complement (NEG x86 instruction). Is $0xFFFFFFFF = ((2^{32})-1)$ or is it -1? You can check your debugger (in-depth introduction later) to see whether the data type is unsigned (positive) or the type is signed (negative and positive). Note from the table below that zero has some redundancy as is represented by multiple symbols in a couple of methods.

For our purposes and keeping in mind the C data types, the data type `char` equals 1 byte, `short` equals 2 bytes, `long` equals 4 bytes, `double` equals 8 bytes, `sbyte` is still a byte (8 bits) with the data range effectively halved, and the MSB now represents the minus sign; `int` equals 4 bytes, `word` equals 2 bytes, `dword` equals 4 bytes, and `qword` equals 8 bytes.

For the C types, you can write a simple program with the lines:

```c
#include <stdio.h>
int main() {
    printf("%d", sizeof(double));
    return 0;
}
```
Insert your data type of choice inside the `sizeof()` operator.

<table>
<thead>
<tr>
<th>Sign and magnitude</th>
<th>One’s complement</th>
<th>Two’s complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
<td>0111</td>
<td>0111</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>1111</td>
</tr>
</tbody>
</table>

Binary addition and subtraction of unsigned numbers is another curious segment. When you add 1 + 1 in decimal, you have the symbol 2 to denote two entities or objects or values, so you can write 2 to the result. However, in binary, the symbol set is similar to decimals only for the 2 values {0, 1}. Hence, to represent larger quantities, you displace the same symbols toward the left to symbolize that quantity. Binary does not use the decimal range, so 2 in binary will be 10, which is not the decimal 10. Is 1 + 1 + 1 = 3? That would be wrong in binary terms because there is no symbol for 3 in binary even if the quantity 3 can be represented validly. So, the resulting value will be the binary symbol sequence 11 and not decimal 11.
Signed numbers have to deal with carry in and carry out comparisons of the MSB position to check for overflow conditions. If the carry in value is the same as the carry out value, there is no overflow; however, if there is a discrepancy, there is an overflow. This is essential for the proper representation of signed data types and addition and subtraction between these types. This is a simple XOR (please read more on gates in the sections later on in this chapter) such as comparison for internal circuitry, which is a much more streamlined affair than the other error-checking solutions. There is an area in the microprocessor to check for conditions such as this during calculations: the EFLAGS register and the OF or Overflow Flag bit, which is set whenever there is an overflow.

### A signed data type overflow conditions table

Let us delve into signed data types and overflow conditions, which can be perused succinctly in the following table:

<table>
<thead>
<tr>
<th>Carry-In</th>
<th>Carry-Out</th>
<th>Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

If there is a carry out at the MSB with no carry in, then there is an overflow. If there is a carry in to the MSB with no carry out, there is an overflow.

For instance:

\[-7 \, + \, (-7) = -14\]

```
  11111001
+ 11111001=
  111110010
```

The carry that was getting into the MSB was \((1 + 1 + 1 = 11, \text{ so } 1 \text{ as carry})\).

The carry out is 1 as well, which will be discarded. However, they are both the same so there has been no overflow and the result is valid as negative 14. You can check it as NOT (bitwise inversion) \((11110010) = 13 \, (0000 \, 1101)\), and add 1 to get 14. It's the 2's complement of 14. Since the MSB is set, the number is a signed data type negative number, which adheres to the representation requirements.
Take another example:

\[
\begin{align*}
1100\ 0000 & \quad (192) \\
1011\ 0001 & \quad (177) \\
\text{ (+) } &= \quad (1)0111\ 0001
\end{align*}
\]

This evaluates to 369, which is larger than the data type range of a byte, which is 256. Hence, we can assume that taking the numbers as unsigned is an error.

However, if we take the type as the signed type:

- The binary pattern is a 2’s complement of 64 decimals as \([\text{NOT (1100 0000)} + 1] = 64\)
- The second number is also taken as a 2’s complement of 79 \([\text{NOT(1011 0001)} + 1] = 79\)
- Taken as signed numbers, we get the correct value as \((-64) + (-79) = 113\), a positive signed number
- As a signed type, the byte will have 127 as the largest positive number and -128 as the largest negative number

Remember that a rollover effect happens if the largest number on either side is reached during the increment. To reach 127 as the largest permissible value in a byte, 63 units are required to be added. After that, from -128 onward, the range is traversed backward toward 0 at the center of the signed range number line. From 79, subtract 63 to get 16 units of increments remaining. Go back that many steps from -128 to reach -113. This is the correct answer within the range.

This same process occurs for larger signed data types as well as for byte lengths such as WORD, DWORD, and QWORD.

A better way to understand negative representation is the simple mathematical result of adding a positive and a negative number of the same magnitude. \(5 + (-5) = 0\). So, you can ask the question: what number when added to a positive number gives 0? This will be key to understanding the negative number representation and its myriad forms of optimized notation systems, and their pros and cons.
Say, we take the decimal 5 and convert it to its binary notation, 0101.

The 1 that is carried at the end is discarded as the requisite value is already obtained and is an overflow for the current data type that can be taken as a disposable artifact for our purposes.

So, we get 1011 as negative 5 as a result. As a positive number, the value is 11. However that is only for the unsigned data type. For signed types, the type data ranges are bifurcated into two parts: positive and negative bit patterns. Note another feature of this result. If you remove 1 from the LSB, you essentially get the 1’s complement of the original value. 5 = 0101 and the (result - 1) = 1010. Does that look like an inversion? Yes, it does. Now, the final result itself is the 1’s complement plus 1. If you look at the bit patterns, you essentially are doing a NOT operation and a (NOT + 1) operation. x86 microprocessors provide instructions that can work at a bitwise level with NOT and NEG instructions. So now, negative values can be computed and represented logically instead of numerically for every number that falls within the range of a data type. However, 2’s complement is the best method currently as 1 does not have to be added and subtraction is simpler, as well as not dealing with positive and negative zeroes. This saves CPU time and additional circuitry design specifically for negative numbers, so the benefit of using the same addition circuitry (ALU) for both addition and subtraction (negation and addition) is very evident.

<table>
<thead>
<tr>
<th>0101 (Minuend)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ ??? (Subtrahend) =</td>
</tr>
<tr>
<td>0000 (Difference)</td>
</tr>
</tbody>
</table>

**STEP 1:**

| 0101 |
| + ??? |
| = 0 (with 1 carry for 10) |

**STEP 2:**

| 0101 (1 previous carry) |
| + ?? 11 |
| = 00 (1 new carry for 10) |

**STEP 3:**

| 0101 (1 old carry +1 =)10 |
| + 7011 |
| = 000 (1 new carry for 10) |

**STEP 4:**

| 0101 |
| + 1011 |
| = (1) 0000 |
We will delve more into other number representation schemes (real numbers/fixed and floating point numbers), BCD, assembly programming, deciphering disassembly, and arithmetic in the coming chapters.

**Boolean logic and bit masks**

Boolean logic can be thought of as a symbolic model that borrows from both mathematics and philosophy to understand, emulate, quantify, and implement specific human thought processes. This scheme was invented by George Boole, an Irish mathematician in the 1800s, in his seminal paper *The Laws of Thought*. George Boole was the first person to come up with a workable methodology to harness the process of human logic in a mathematical framework.

The best way in which Boolean logic can be expressed in electrical and electronic engineering terms would be the series (more battery power) and parallel (longer battery life and reduced current) circuits.

An AND gate can be constructed as a simple closed series circuit that consists of two switches, a battery, and one bulb/LED. Only if both switches are closed will the bulb light up.

An OR gate can be constructed out of the same building blocks as the previous circuit, except that the switches are kept in parallel. Toggling any one of the switches or both at the same time will light the bulb up. The switches can be taken as the inputs to the gates.

Another invention called the relay switch uses magnetism and mechanics to toggle switches on and off without human intervention. Later on, with the invention of semiconductor devices such as the transistor, the need for mechanical parts was removed and they act as electronic switches that perform the same function with more durability and reliability (unlike obsolete vacuum tubes as the prior intermediary technology).

For our purposes, the most important logical operators are AND, OR, XOR, and NOT.

AND and OR are dyadic operators. NOT is a monadic operator.

AND takes two operands and produces a 1, only if both inputs are 1.
OR takes two (or more) operands and produces a 1 if either or both inputs are 1. Ever wonder how bit flags during programming are OR’d, one after the other? They are individual bit positions, and hence, an OR operation can be used to combine multiple bit flags.

Both AND and OR produce 0 for both inputs of 0.

NOT takes a single input and inverts it. If the input is 1, then the output is 0 and vice versa.

XOR (ex-or) takes two operands and produces a 1 only if either of the inputs is 1 and the other is 0. If both inputs are 1 or 0, the output is 0.

A curious feature of XOR is that XOR'ing two similar values produces a 0.

XOR'ing the output back with either input produces the other input. C XOR A = B & C XOR B = A, if A XOR B = C.

XOR is used in assembly instructions to set a register to zero in order to initialize values to a variable and is used for basic encryption and error checking.

A truth table for each operator provides a tabular view of the inputs and outputs of each logic gate.

<table>
<thead>
<tr>
<th>AND</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Can you build the truth tables for the other Boolean operators?

**Bit masking**

Using AND and OR, we can extract or manipulate certain bit positions; this will be instrumental in understanding the process of bit masking. A bit mask is essentially a bit pattern applied in tandem with one of the logical operators to affect the target bit pattern so that certain bit positions are either set to 1 or 0. Think of masks as a filter to extract or block bit values at a particular position. This has various uses such as working on a bit or nibble level as the x86 instruction set does not allow bit-level manipulation directly, unless you are using one of the bitwise operators such as SHR or SHL (which are shifts made on the bit pattern going right or left a certain number of positions as required and the opposite end being padded with zeroes) among others.
Bit masking can also be used to simplify certain programming and conversion tasks such as uppercase characters to lowercase, which can be done by setting the 6th bit (index 1 to 8th bit) of the ASCII value to 1 (lowercase); you are encouraged to derive this on your own as an exercise. Both uppercase and lowercase codes differ only in the 6th bit. Of course, in Windows, everything is Unicode, so this is a thing of the recent past but serves as a good example. Visit https://msdn.microsoft.com/en-us/library/windows/desktop/dd374081%28v=vs.85%29.aspx to learn more about it. More importantly, you will find masking of memory addresses to a multiple of the memory alignment size (1000H or 4K) as a common occurrence in various algorithms and even in malware disassembly.

Since AND'ing any pattern with 0 will result in 0, AND can be used to zero out a bit pattern. OR'ing any bit pattern with 1 will let that value pass through. This can be used as a bit setter. Say 1110 1110 (EEh) AND 1111 0000 (F0h) = 1110 0000 (E0h) and 1110 1110 (EEh) OR 1111 0000 (F0h) = 1111 1110 (FEh). So, to summarize, we can use a bitwise:

- AND for testing/zero masking
- OR for setting
- XOR for toggling (can you figure out why?)

Let us have a short tour of a malware analyst's toolbox before we move onto code constructs and disassembly.

**Breathing in the ephemeral realm**

Ideally, how you approach malware analysis from the perspective of disassembly code is largely dependent on your required objectives. While complete code coverage is certainly possible to a good degree, it is not always practical; hence, you have make a judgment call after you reach a point of diminishing returns, wherein exhausting the available resources will not yield a significant value any further. I believe that the three tenets of successful malware analysis include pattern recognition, the process of elimination, and cross-checking the available information. Concisely, it is a problem solving mindset with solid coding skills. Deciphering dead listings or raw disassembly text without executing the binary is one of the staples in any given malware analysis session and certainly has an l33t air to it as well as that of being a legacy method of binary code analyses in the earlier days of malware research. Of course, times have evolved and analysis automation is the order of the day, which given the quantity and quality of malware in vogue is a recommended process with mixed results. However, if you ever wish to do a dry run of any form of source code, it does not get more involved than this (also, if you enjoy tedium and have masochistic tendencies).
Prior to reading disassembly code, you will also have to do binary reconnaissance in order to facilitate better static analysis. This would mean an analysis of the binary file format (PE/Coff format) in order to detect anomalies and understand its overall structure, also known as header values and section data; take note of the special areas of the binary such as the data directory structures export/import/debug/tls, demarcate overlays in the binary, its record hashes (MD5/SHA1/2), and extract strings among various other procedures. All this activity is more like due diligence of the investigation a priori and will add value to your analysis efforts.

Further, in the ideal dead listing session, runtime data will not be available immediately (you can always use debuggers or virtualization to get runtime data—dynamic analysis, which we will cover in the chapters ahead). This means things such as command-line switches, various input data, decryption and unpacking procedures, and multi-threading may have to be emulated via a judicious state recording of the CPU register values and the simulated process memory data structures. While you can copiously comment and subtext relevant lines of disassembly and watch call graphs in the disassembler, as well as edit function names and depict code as data or vice versa, nothing beats pencil and paper to etch mental models of the execution trace as well as depicting the complex interactions between different program and OS components.

Your tools will only help you with presenting information contained in the binary in various ways. Your mileage depends on your knowledge level, experience, insights, and purpose. That said, malware analysis is not rocket science but rather an esoteric craft (software reverse engineering)—both art and science—and you do not need to be a guru to avail of this skill set, as anyone with time, patience, and practice can become a very capable analyst (and anything else, for that matter).

You will cover the following topics in this chapter, which will enable you to perform a static analysis with confidence.

- IDA Pro, PE Explorer, and other analysis tools
- Foundations of reverse engineering

Sharpening the scalpel

The regular disassembler is a static analysis software tool that performs many different processes and extracts information out of a binary executable. It parses the binary executable, takes apart the individual sections, and presents a list of annotated assembly code from the binary string of opcodes embedded inside the executable.
Additional embellishments arrange relevant data such as symbolic function and variable names (if present), stack frames and variable lists, common data structures, strings, and import and export function jump lists. There are two primary algorithms that are implemented in a disassembler:

- Linear Sweep (WinDbg, Win32Dasm, Sourcer)
- Recursive Traversal (OllyDbg, IDA Pro)

Linear sweep processes a binary executable by navigating to the code segment and reading the binary strings as a linear sequence of bytes, which are decoded according to a table of opcodes for a specific instruction set, much like a mapping process with an incrementing position counter to keep track of the next byte sequence to decode. The primary caveat is that because linear disassembly assumes the code section as pristine without being tainted by data elements, additional code constructs such as unreachable code, code interspersed with data (switch tables and function pointer tables), opaque predicates (fixed-value false conditional expressions), and stack frame pointer resolution cannot be done with confidence as cross references such as function call statements are not maintained. Thus, complicated machine code sequences can confuse the disassembly and result in junk code. However, code coverage is a feature that can be availed of when necessary.

Recursion in computer science might remind you of mathematical induction and a growing stack-based nested function call sequence of a function calling itself. The recursive function requires a terminating condition in order to halt a repeated procedure for input values by calling itself repeatedly till the terminating condition is met. However, recursive traversal disassembly algorithm is a relatively complex undertaking that can be implemented in numerous ways. As a generic approach, all conditional (jnz/jxx) and unconditional code constructs (jmp/call) are taken into account and the control flow is traversed wherein the pseudo custom C data structure is as follows:

```c
typedef struct _instruction_metadata {
    unsigned int *instr_offset; /* instruction offset in executable or eip if emulated */
    unsigned short op_length; /* processed opcode length */
    unsigned int *dest_address;
    char array [op_length]; /* opcode sequence */
    unsigned int *return_address; /* for address of next instruction after call */
    /* also current offset + opcode size */
    /* data structure representing internal parameters required by the disassembler */
    MetaData meta;
} INS_META;
```
This structure is saved in the disassembler's internal data structures as **branch lists** (also known as jump list – which is confirmed code instructions or return list – which is yet to be identified addresses-code/data/tables) for resolving all control flow pathways. After each linear control path analysis pass, another address is retrieved from the branch list and the evaluate list, and the disassembly process resumes till both lists are empty. This list is re-processed in **analysis passes**, and cross references are drawn from the prior analysis, resulting in a much more coherent disassembly listing. Quite a bit of heuristics (for instance, compiler type-based assembly code templates and EBP or ESP-based stack frames) and code instructions vs. data statistical analyses strive to make this a more reliable process. The disassembly also annotates the disassembly code accordingly with identifiers and labels, and could even provide with disassembler-generated comments for familiar code sequences, to get the final code listing. Binary code disassembly can be an intractable problem, particularly if requiring user input or external data and with no named symbols in the executable, and things such as variable and function names are lost. While the disassembly will provide a lot of insight by presenting the code in the best possible light, whatever is remaining will have to be semantically reconstructed from the disassembly manually or using advanced algorithm-based code analysis tools. We will cover the standard disassemblers in vogue, as well as code analysis tools aimed for high-quality reconstruction and analysis. Because of the complexity involved, recursive traversal is more time consuming than linear sweep but more accurate and resilient to the issues that can halt the linear sweep process, and therefore the algorithm of choice for our purposes.

**Performing binary reconnaissance**

The PE format is the executable binary format in Windows. The overall structure of a PE file is shown in the exhibit; the PE file has a bunch of headers, which are metadata for the Windows loader to help load the image to process memory. The MZ or DOS header starts with the MZ or 0x4D 0x5A magic number. The 4-byte value at offset 0x3C from the offset 0x0 of the MZ header gives the location of the start of the PE header, which has the signature 'PE\0\0' or 0x50 0x45 0x0 0x0. The PE header contains the optional header, which is a legacy term and is certainly not optional. Thereafter, the section header begins, which contains the metadata describing the sections and their properties—section name, raw and virtual size, and address and section characteristics. Thereafter, the sections themselves are linearly appended, one after the other.
The following is an excerpt from the COFF specifications:

The header data structures are as follows: Several underground articles also exist that do an admirable job of documenting the PE format; Goppit's PE tutorial, B. Luevelsmeyer's tutorial, and Kris Kaspersky's *Hacker Debugging and Hacker Disassembling* are also good references. Those who are interested in the golden age of Windows reverse engineering must search for + ORC's legacy on the Internet.

**Downloading the example code**

You can download the example code files from your account at [http://www.packtpub.com](http://www.packtpub.com) for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files e-mailed directly to you.
Scanning malware on the web

If you can afford to submit a malware sample (say, make it public) or a hash of the sample to online scanners, you should be able to get a detection report and submit the sample for posterity, particularly if it is undetected and actually malware. Virustotal.com and Jotti.com as online scanners, and Anubis, Minibis, Cuckoo, BitBlaze, Comodo, ThreatExpert, Zero Wine, and BSA-Buster Sandbox Analyzer (potential abandonware though works for user mode malware that can run on XP and Win 7) are dependable among the myriad of scanners that fulfill this role. Installed antivirus products on your main host system or virtual machines can also be availed of to get the preliminary screening done; they have to be updated prior to scanning to ensure that they do not miss available signatures. Most have On-Demand Scanning, which is manually invoking the scanning process on a user-selected file, usually through shell integration (also known as right-click context menu item) in Windows. This is different from On-Access Scanning, which intercepts I/O calls and network activity via kernel filesystem and network filter drivers in all running OS processes, as well as removable media and network downloads to the filesystem, and proceeds with scanning them for malicious code, letting them continue if they are benign or else proceeding with termination, quarantine, or deletion of any malware present. Much of the toolsets that we will implement will focus on the PE/Coff format. Streamlining your toolkit must be an ongoing pursuit in addition to creating custom scripts and tools as and when needed; gear lust is not as important as knowing how to use the ones that you have inside out. Even if all the tools are taken away from an analyst, if he/she has a debugger, disassembler, hex editor, and development IDE (arguably, the only tool required as others can be made from this given enough time and motivation (and brains)), he or she can still fulfill their role. Before you examine the PE format, let us look at the battery of tools that we can incorporate in our daily analysis sessions for binary reconnaissance. Make sure that you have the most recent and updated versions of the following tools as this software can and will contain vulnerabilities that can be exploited, particularly given the context of malware analysis. The PE format is very well documented in MSDN: https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx

Further, a good graphic of the PE format is available at https://raw.githubusercontent.com/corkami/pics/master/PE101.png
Getting a great view with PEView

PEView exudes simplicity and a good GUI design as compared to other click-and-browse tools while being a very robust format parser for every executable that you can throw at it. The tree control to the left categorically breaks up a PE binary into its constituents. The right view displays the hex dump or the section and header attributes. In the figure, the .text code section is selected with the hexdump view towards the right pane:
Know the ins and outs with PEInsider

PEInsider from Cerbero is a more recent alternative to the Explorer Suite (CFF Explorer), its earlier offering. This is one of the more thoughtful designs with highly informative context-based displays. The overall features are similar to those of PEView.

Identifying with PEiD

PEiD is a now defunct binary utility that is still immensely useful in detecting packers, compressors, and compiler/linkers, and has a slew of PE format-related features, including a concise GUI display of the most pertinent attributes of the PE executable, information gathered from the PE header/Optional Header/Data Directories/Section Headers, a task viewer to enumerate and manipulate running processes, including taking memory dumps and a dependency list, shell integration (right-click on Explorer.exe), and entropy calculation. A mini hex viewer and basic disassembler with control flow navigation (jmp/call) is directly linked with the PE section(s) view context menu along with section header attributes, which enables inbuilt navigation to the binary file for investigation. Signatures for packers and similar utilities are compiled in a text file called userdb.txt. You can add your own custom signatures to the database.
The two main scanning modes are **normal** and **deep**. Normal mode scans the original entry point (OEP) of the executable for a match in the database signatures. Deep mode scans the section containing the OEP for a more accurate match. There is a **hardcore** mode that scans the entire file and thus, is used as a last resort. Finding the OEP can be a challenge many a times, if the packer is unknown.

Some useful plugins are included such as KANAL plugin for crypto analysis and the Generic OEP finder.

In the following image, the packer/compiler display label shows **PECompact 2.x -> Jeremy Collake**, which is the detection of the packer used in the input sample, PEiD itself in this case.

Obfuscated files and packed/compressed files are somewhat different implementations of the process of concealing something in the obfuscated binary that aims to confuse the human analyst as well as automatic code analyzers, while retaining the functionality of the original binary. The various obfuscation methods include source code- or binary code-specific manipulations, such as string mangling and the removal of source code variables and function identifiers to something seemingly random and redundant, thus increasing the complexity of the control flow analysis. Packing and compression focuses on code compaction and the reduction of the total binary file image footprint. This procedure increases the entropy and hence the obfuscating binary, to a certain extent. The PE file header values and sections are non-standard (manipulation of linker-generated headers and sections) and packer-specific; they rebuild the compressed import and export tables in the memory, which is usually done by an unpacking stub.
Some code armoring is also done using virtualization, and an intermediate representation of the original assembly instructions that are mapped at runtime and JIT compiled by the execution engine. Care must be taken not to invoke the OEP finder plugins and the other unpacking plugins for PEiD, as doing so will execute the malware on the system and can possibly infect it during the analysis.

**Walking on frozen terrain with DeepFreeze**

*DeepFreeze* from Faronics is a great utility that has been around for quite some time now. It is excellent for use with the main standalone host system, and particularly, with a virtualization setup using VMware or VirtualBox to preserve system integrity. The baseline state is saved when you activate it from the notification bar in Windows, and post installation, execution (deliberate or accidental), and analysis of malware and the related packet captures, dropped files, and memory dumps, you can simply revert to your original baseline as many times as you like. Uninstalling can be a bit tricky for some installations (BIOS settings have to be manipulated), and be sure to remember the login password because even uninstalling from the Control Panel will not proceed if you lose the password and you could be stuck with a frozen machine that persists nothing else.

**Meeting the rex of HexEditors**

Hex editors are an essential utility to work with binary executables as they enable an as-is unbiased view of the file format that is as is and not tainted by any kind of parsing logic of a format viewer. This enables you to work at the byte level and build entirely new binary executables from scratch. Utilities are available that allow ease of editing on PE files built by motivated individuals over the course of research and coding cycles, which makes this look easy; however, when required, a well-featured hex editor gives unfettered access to working manually with import and export tables, PE headers, sections, overlays, wire captures, and memory dumps, among other assets. It is simply a hexadecimal view of a binary file, expressed as a binary string. Hex editors vary in the gamut of features provided out of the box; some of the more important ones include multiple file views, data structure templates, data type viewing, entropy calculation, hashing tools, strings search, offset navigation, bookmarks, and color mapping, file statistics, Boolean operations, encryption/decryption, support for large files/alternate data streams/filesystems, live acquisition, forensics utilities, and even a disassembler.
**Hex workshop** is a very well-featured hex editor with many of the features discussed. 010 Editor has a great interface with flexible templates for data structures. Flex Hex supports large files and alternate data streams. **WinHex** is more of a forensics-focused hex editor with RAM editing and MBR/hard disk-focused features.

**Digesting string theory with strings**

**Sysinternals Suite** has many useful utilities, and the `strings.exe` command-line utility does this one task very well. It takes the target file or folder as an input.

The `-a` and `-u` switches can be used to display only ASCII or UNICODE as both are displayed by default. The `-o` switch appends the offset of each detected string in the file. `-s` can be used to recursively traverse a parent folder. The string length of 3 is set by default and can be changed using the `-n <length>` switch.
In the following figure, we can see the familiar fileoffset:string combination display of the MZ header and the standard section names .text, .rdata, .data, and .rsrc with the rest resembling junk. Version strings, API names, URIs, URLs, IP addresses, password lists, FTP login strings, HTTP GET requests, mutex names, HTML pages, IFrame tags, and embedded scripts are among some of the categorical strings that you can look out for during malware analysis.

Bintext from FoundStone Inc. is a GUI strings tool and is another solid complement to the Sysinternal strings utility.

For runtime strings analysis and image verification (Image tab | Verify button) of digital signing by certification root authority, you can use Process Explorer from Sysinternals. Although an execution monitoring tool, this tool has numerous features that can be utilized for malware analysis. Double-click the process name, navigate to the Strings tab, and choose the Memory radio button to see the strings in the process memory.
If the malware is packed or compressed, the Image strings and Memory strings will be significantly different as the memory-unpacked regions have their packed strings unfurled.
The ASCII chart is given next for reference. You would read it like a grid value from
the row and append the column value to obtain the resulting decimal value. The
ASCII character '1' is read from row 4, and the column value is 9. You concatenate 4
and 9 to get 49, which is also 0x31.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Null</td>
<td>Tel</td>
<td>Backspace</td>
<td>Horizontal Tab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Und</td>
<td>Vertical Tab</td>
<td>Form Feed</td>
<td>Carriage Return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(Space)</td>
<td>Cancel</td>
<td>Escape</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>!</td>
<td>*</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>^</td>
<td>&amp;</td>
<td>_</td>
<td>'</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(</td>
<td>)</td>
<td>+</td>
<td>-</td>
<td>.</td>
<td>/</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>8</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>10</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
</tr>
<tr>
<td>11</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
</tr>
<tr>
<td>12</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>(</td>
<td>)</td>
<td>_</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASCII values are 7-bit encoded numeric codes assigned to symbols, which consist
of natural numbers, upper case and lower case letters of the English alphabet,
punctuation symbols, carriage return, line feed, null, backspace, and other special
characters. Because 7 bits are used, there are 2^7 or 128 characters that can be used.
ASCII provides for the first 128 characters of Unicode. Unicode uses 16 bits, and
hence, provides for more than 65,000 numeric codes, out of which about 40,000 are
already used, which leaves a lot of codes for future use.
Hashish, pot, and stashing with hashing tools

Hashing utilities are useful for identifying and watermarking files and malware samples for further processing and for posterity. The MD5 and SHA-1/2 variants are the most commonly used algorithms. FileAlyzer 2 has a comprehensive hashing feature (the **Hashes** tab) and an interesting set of features for preliminary binary and text file (scripts and html) investigation, including network support for online malware scanners such as [https://www.virustotal.com/](https://www.virustotal.com/). Shell integration means that any file can be analyzed from the Windows Explorer context menu, which makes it the first tool that you may ideally use for initial triage. It packs quite a bit, including a disassembler, a hex viewer, packer/compiler detection, support for archive files, ssdeep hashing using imported DLLs (**ssdeep.dll**), and Clam AV scanning (**libclamav.dll**).

<table>
<thead>
<tr>
<th>File Edit Window Report Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashes</td>
</tr>
<tr>
<td>Disassembler</td>
</tr>
<tr>
<td>General</td>
</tr>
</tbody>
</table>

| Location: | C:\Users\user\Desktop\ |
| Size: | 30720 | 0000000000007800 |
| Version: | |

**CRC-32:** 4035208A  
**MD5:** 5D9EDF4C5212ED0AAB863B61995C57CD  
**SHA-1:** 2A5538E7BE1774376A0D9393481D42C4ED3F394C4B

| Read only | Directory | Microsoft Visual C++ 8.0 [Debug] |
| Hidden | Archive | |
| System file | Symbolic link | |

**Creation:** Saturday, May 9, 2015 4:54:42 PM  
**Last access:** Saturday, May 9, 2015 4:54:42 PM  
**Last write:** Saturday, May 9, 2015 4:54:42 PM  
**Creation (UTC):** Saturday, May 9, 2015 11:24:42 AM  
**Last access (UTC):** Saturday, May 9, 2015 11:24:42 AM  
**Last write (UTC):** Saturday, May 9, 2015 11:24:42 AM
Nirsoft’s (Nir Sofer) **HashMyFiles** utility is an excellent Windows-based GUI hashing tool. It takes a file or a folder as input and lists out in columns hashes for MD5, SHA-1, CRC32, SHA-256, SHA-512, and SHA-384. It also displays the created and modified times, the full path of the file, file size, version strings, file extensions, identical files, file attributes, and https://www.virustotal.com/submission.

Hashing tools can be used to generate malware database hash lists, as well as for checking the integrity of the existing binaries. Hashing also plays an important role in antivirus signature creation. During and after analyses, you would be ideally using a hex editor to create checksums and hashes of byte regions with parameters such as the number of bytes to hash, as well as the start offset and with additional parameters from the OEP or some offset of it or backwards into a file or from the last byte of a file and hashes of overlays, among other options. Once you have the required hashes, you would be customizing and compiling them for each binary by using vendor-specific **VDL (Virus Definition Language)** or Signature SDK, which exposes a set of APIs that the antimalware engine utilizes for detection during scanning. Quite a few of the vendors have internal and networked point-and-click interface systems for the analyst exposing features of the sample(s) and generating hashes and processing them directly from the sample queues without even having to download the sample binaries for a local system analysis, unless mandated. The relatively simple 1:1 static signatures look something like the following:

```
Trojan "Win32/Agent.AB" // malware is a trojan, nomenclature of Agent and a variant name of AB
{
  entrypoint(0x15B8);   //entrypoint in the raw file image
  HashEntrypoint(0x0,0x4B0,0x127C099B); //hash of OEP dropout till 0x4B0 of malicious data
  HashFileStart(0x3E15,0x1BE0,0x00E44360); //data island in code section + related code
}
```

A moving window and wildcard-based signature looks like the following:

```
Name:Dropper.Malware2.AA
{
  $1 = 4A 4F 38 34 30 31 50 52 49 4E 43 50 48 41 53 54 5C 54 65 6D 70 5C
  $2 = 50 FF 90 58 02 00 00 59 33 C0 C3 55 8B EC 81 EC 04 [4-6] 8B 75 08 57 8D BE F8 04 00 00 57 33 DB 53 6A 04 FF 96 34 03 ?? ?? 85 C0 0F 85 B4 00 00 00
```
Here, the pattern is in hex bytes; the square brackets denote that for the length of the next 89 bytes, if you find the value 30h, then you can continue with the rest of the signature. The ?? wildcard means that any byte value can be present after EBh and before 40h. Since the OEP is not specified, the OEP is searched first and then the beginning of the code section and then the top and tail of the file, including all sections and overlays (except the code section that is not scanned again). The resource malware-specific icon asset is checked for a checksum match.

Polymorphic malware is checked for the decryption stub and the decrypted malware code and other file properties to enable robust detection without using brute force on the key space or doing exactly that for oligomorphic malware that has a fixed or a feasibly finite number of decryption keys.

These basic signatures are then recompiled to a performance-efficient custom binary format before they are fed to the antimalware signature database. They are also made modular so that live updates are possible. The other variant is generic signatures wherein heuristics and data mining algorithms are implemented to capture the essence of a malware family or generation or fingerprint a new one if it is a variant. This requires creating a failsafe set of conditions in the format that is specified by the generic detection engine, usually as a detection script that returns a positive detection if all or most of the conditions are met. This is usually a more involved effort and requires judicious testing for false positives and false negatives till the point of diminishing returns. API sequence profiling and instruction opcode statistical analysis are some of the methods that can be used to provide inputs for generic signatures.

To get an idea of how antivirus products can be analyzed, have a look at the following links:

- https://lock.cmpxchg8b.com/sophailv2.pdf

Here is a paper on fuzzy hash: http://jessekorblum.com/presentations/cdfs107.pdf.
Getting resourceful with XNResource Editor

XNResourceEditor is a well-featured and easy-to-use resource editor utility for executable files. The resources are a set of binary assets that are compiled using a resource compiler to the format expected by the PE specification, which the linker finally integrates into the resulting executable. Usually the .rsrc section in the executable contains the compiled resources. The Bitmap section displays bitmaps that are used by the executable; the Dialog section displays dialog items implemented in the executable, which could be the main interface template; the Version strings contain properties such as ProductVersion, ProductName, FileVersion, FileDescription, LegalCopyright, LegalTrademarks, and CompanyName, which could be used to add detection logic post analysis; String Table contains null-terminated Unicode strings (Windows is fully Unicode, although ASCII text can also be present in the binary); Cursor Group contains cursor files; and Icon Group contains icon files.
Many malware binaries contain junk text- or malware-specific identifiers in the Version section and the String Table section. Resource sections can contain malicious code as they can be any binary asset, and hence, are important for the purposes of investigation. Further, even if an executable is not executing or is corrupted in the code, possibly post re-infection or an error in the unpacking algorithm, the OEP might be patched badly or might require a condition for redirection among a host of other reasons. If the resource section is relatively unaffected, you can still investigate the binary prima facie with a resource editor as the dialogs and strings, as well as the interface layout and icons, can reveal a lot about the binary in question. You can then use the Internet to gather more information about the resource assets. The assets themselves can be examined for validity and embedded shellcode or malware. Anti-malware signatures for fake antivirus and spyware (where the resource icons, strings, bitmaps, version strings, and for that matter, any confirmed malicious asset in the resource section) can be included in composite malware signatures and in the detection logic post the extraction of resources.

**Too much leech with Dependency Walker**

`Depends.exe`, or Dependency Walker, is a very thorough tool for providing detailed listings of all dependencies that are statically linked and dynamically called via imports or delay-load imports.

In the following image, the sample `bcb6kg.exe` file is set as input (drag and drop). We see the list of imported DLLs on the left-most pane tree control. If you select any entry in the list, such as `Kernel32.dll`, the adjacent top-right pane contains the list of functions imported from `Kernel32.dll`. The pane just below it will display the complete set of functions exported by `Kernel32.dll`, which depending on the context, might not be too useful as of now. The pane above the bottom pane displays the binary information of the modules that will be loaded by the sample executable. The bottom pane displays error messages and the log output of the activities.
The runtime profiling feature *F7* ensures that dependencies that cannot be resolved via a binary static analysis will be integrated into the report. Of course, certain libraries might not be invoked without external input or user intervention, and in such a case, it will not be detected, but in most cases, it will do a reasonably good job:

**Getting dumped by Dumpbin**

This is the Swiss army knife for everything PE from Microsoft. It comes with the MASM32 SDK as well as most of the developer SDKs by Microsoft, including Visual Studio installations (via Visual Studio Command Prompt).
Simply type DUMPBIN /ALL <filename> to dump everything about a PE file. You could append it to a text file for ease of recall as the console buffer will run out.

For simple automation, you could type the following in cmd.exe on a folder of binary samples for batch processing (replace the %i variable with %%i for a .bat batch file).

FOR %i in (*.exe *.dll) do dumpbin /imports %i >> imports.txt

Dumpbin.exe has to be in the current folder or configured in the environment variables. The preceding command enumerates all the .exe and .dll files, invokes dumpbin with the /imports switch, and appends the output to imports.txt in the current folder. You can replace the switches accordingly. The following screenshot shows how the imports are reported in Dumpbin with the virtual addresses (with respect to the image base of the executable) and function name hint values as well as the function name strings in their own columns. In case the function names are not present, the function ordinals are used instead, which are just numbers (from #1 and not 0).
The `/DISASM` option produces an acceptable disassembly of the code with the
`:bytes` (default) or `:nobytes` option to display the hex opcodes or just the assembly
mnemonic listings; you need to type the following line of code to display just the
assembly listing:

```
dumpbin /diasm:nobytes<filename>
```

With an array of essential tools at your disposal, you may think that it would be
redundant to have tools that can implement possibly much of the available toolset.
In the good old days, reverse engineering started with plain Jane developmental
tools such as basic debuggers, printouts, and paper and pencil. Notwithstanding the
culture of homegrown tools by the underground elite, as the industry developed
over the years, specialized tools (free and commercial) started being developed as a
result of R&D. We will discuss two such tools for our purposes of a static analysis of
a binary executable and incorporate disassembly analyses in the equation:

- **PE Explorer**: This is a lightweight doppelganger of IDA Pro with a lower
  price tag but having a similar feature set and possibly a more integrated
  feature set, while not as extensive as IDA Pro.

  A relative new comer with good looking prospects is available at [https://www.relyze.com](https://www.relyze.com), which provides much of the features you would expect
  from IDA Pro albeit in a more streamlined interface.

  You can also check out Hopper disassembler if you are reverse engineering
  on MacOSX which also does PE files and has a very unique well designed
  feel to it. Visit [http://hopperapp.com](http://hopperapp.com) for more.

- **IDA Pro**: The industry standard for binary reverse engineering. We will cover
  scenarios and the multitude ways of analyzing native binaries by using it.

**Exploring the universe of binaries on PE Explorer**

PE Explorer from Heaventools (Germany/Canada) is a well-featured toolkit for a
static analysis of the following PE file format extensions in Windows—EXE, DLL,
SYS, DRV, MSSTYLE, CPL, OCX, BPL, DPL, SCR, and FLT— and Windows CE
binaries. The GUI is intuitive and not at all complicated. The approach here is that
every aspect of a PE binary has its own separate view. The price tag of $129 offsets
any perceived deficiencies as the disassembler is very capable and the exploded view
provided of a PE file is second to none. However, there is no debugger and the code
cannot be edited (you can use an external hex editor), so dynamic analysis is not an
option, which in the right situation, maybe exactly what you need. The basic editing
features are only provided for the header flags and timestamps.
The HEADERS INFO, DATA DIRECTORIES, and SECTION HEADERS toolbar items (the View menu) display each item in a tabular arrangement. In the figure, notice the value in the Real Image Checksum textbox and the Checksum field value in the right-most pane of 0x0002BC86. This is the link checksum value inserted by the linker; the real checksum is calculated during the load time of DLLs or system drivers by the Windows loader to check memory integrity. In general, any discrepancies result in discarding the particular instance.
When the **Editor** (Ctrl + Shift + P) button in the **Characteristics** column is clicked on, an edit dialog enumerates the flags for this field.

![Characteristics Dialog](image)

All the flag values are OR'ed (each value is different, so the binary patterns just fall in place with respect to their respective position in order to resemble a composite binary pattern) to get the final value in hexadecimals in order to communicate to the Windows loader of the required values in the binary header field. Some values are of special importance to us for malware analysis; **0x2000** signifies that the file is a dynamic link library (DLL), and conversely, **0x0002** signifies that the file is an executable (no unresolved external references), which is an EXE file in this instance. **0x1000** would signify that the file is a system file, such as a driver (.sys). The remaining flags are also important, and they convey the validity of the executable to Windows, such as memory usage of more than 2 GB and swap file usage if the file image is on removable media or the network, among others.

Export tables and import tables are described in a similar fashion. Integrated **Quick Function Syntax Lookup** is a great feature for both learning and investigating standard Windows APIs, instead of spending time with manual lookups. **Authenticode Digital Signature Viewer** is a feature to verify the authenticity of the publisher via a certificate-based digital signature. A very handy **Resource Editor** is also provided.
The disassembler (Ctrl + M, Tools | Disassembler) opens in its own window and overlaps the main interface, which can be toggled back anytime.

Name List to the right provides a list of labeled addresses (including conditional and unconditional branching destinations, function prologues, named data, and string references) by the disassembler, with the entry point clearly indicated. Labels can be renamed by pressing N (Edit | Rename Label).

The lower left tabs View 1, View 2, View 3, and View 4 (F6, F7, F8, and F9) provide persistent disassemble views that are independent of the main view and are swappable.

The Strings tab provides a list of detected strings; you can further manipulate strings detection by using the toolbar, using menu items (Edit | Mark as String/Pascal String/Long Pascal String/Unicode), or pressing S, A, L, or U to activate each of them.

Code can be manually marked in the assembly listing by pressing 'C.' Dwords and offsets can be marked by pressing D and O, respectively.
Comments can be entered by pressing ;.

The unprocessed data tab displays some blocks of data that do not have a reference to a procedure.

The main disassembly view is towards the top-left. A nice feature in this view is the provision for an immediate adjustment of the space between each assembly line (Ins and Del) and the number of opcodes per line (Shift + Ins and Shift + Del).

Navigation is really simple. Branching addresses can be navigated by selecting the relevant line and pressing Enter. For instructions with a second operand destination address, press Ctrl + Enter. Going back to a previous address requires pressing Esc, and to visit a particular address, you have press Ctrl + G and type the address in the hexadecimal format.

Subroutines that might have references can be listed in a pop-up window by selecting the starting address of the procedure and pressing R (Search | References). The list can then be traversed by double-clicking on each listed address.

Automatic unpacking is done for UPX, NSPACK, and WinUPack, and the file can be saved unpacked to the file system.

The disassembler options (View | Disassembler Options) provide with a list of instruction sets to disassemble for. The checked Auto Rescan option and Auto Rescan count value are fine at default values, but for complicated binaries, they may require more passes. The number of displayed opcodes can be set to a default value.
The Advanced tab provides for settings that are fine as default.

A dependency scanner (Ctrl + N) hierarchically lists out the external modules and library files that are requisite to a successful execution of the primary binary.

---

**Getting to know IDA Pro**

With the tools that we have covered thus far, you must have a good idea of the workflow toolchain required for a static analysis. Let us now introduce ourselves to IDA Pro (The Interactive Disassembler) from Hex-Rays. The IDA Pro Book by Chris Eagle is a solid reference and guide book towards building mastery in IDA Pro and reverse engineering in general. Since there would not be too much use of regurgitating all of the IDA Pro-specific material and given the space constraints, we will go over the often-used features in IDA Pro and build familiarity with this tool.
Upon opening a binary executable in IDA Pro (drag and drop in the Open menu), the **Load a new file** modal dialog pops up:

![Load a new file dialog](image)

The binary format is parsed and identified by IDA Pro, and the correct loader is prompted as a **Portable executable for 80386 (PE) [pe.ldw]**. The binary file option can be used if you are working with a hex dump without a known header. IDA chooses to load only the code section, and if you need to work with the PE headers and resources, choose the manual load option and select **Yes** for every section that loads turn by turn.
IDA Pro has two main views for working with disassembly listings, namely **Text Mode** and **Graph Mode**, both of which can be toggled via the *Spacebar* key. **Graph Overview** is an eagle's eye view of the current graph block. The rest of the tabs of significance include the **Imports and Exports** (when working with DLLs or uncommon EXE files with Exports) view. The **IDAView-A** tab and the **Hex View-A** tab can be synchronized (right-click | Synchronize with IDA View-A) such that selecting a hex offset in the hex view will result in the corresponding disassembly in the IDA view and the converse. Additional IDA views can be created via **View | Open subviews | Disassembly**.
Strings will be listed in a separate view and can be invoked using Ctrl + F12 or via View | Open subviews | Strings. From the Options menu, the ASCII string style dialog (Alt + A) can be invoked, which provides various string interpretation settings.

You can comment the disassembly by pressing ; and typing the comment in the popup text box.
You can redefine the code in the disassembly by pressing $U$ for undefining code, subroutines, or data. Press $C$ for code representation and $D$ for going back to data for the selected regions in the disassembly view to tell IDA Pro to analyze a particular byte sequence as code or as data. You can press $A$ to mark the raw bytes as ASCII strings.

Right-clicking on an operand in the IDA view will enable you to swap the radix of a type from binary ($B$) to decimal or hexadecimal ($H$), and perform a NOT operation or a 2's complement operation on the value. The **Use standard symbolic constant** option opens a dialog where you can choose the named constants from the comprehensive collection available.
Quick view (Ctrl + I) is a nice linear listing of available views in a pop-up dialog through which you can invoke additional views.

The Functions view provides a listing of all detected functions in the binary, along with the function name string, start offset, and length in bytes. A set of flags denotes the type of function call (R/F/L/S/B/T) with L being library functions, which can be either marked for a vulnerability analysis or skipped for a regular malware analysis as your primary goal is the malware payload(s). You can right-click and choose Edit function to open a dialog box with different editable parameters. You can manually set the function as a BP-based frame or an SP-based frame.
The frame pointer delta is for when the stack pointer is not aligned to its frame-based preparation value and is at an offset further from the original stack frame; while IDA Pro does its best to resolve such scenarios, you can amend any errant stack analysis on the basis of your knowledge and analysis of the stack delta value in hexadecimals.

A particular setting to do for a more informative disassembly is to set the number of opcodes for display in Options | General | IDA Options | Disassembly-Number of opcode bytes. 6 is an optimum value and covers most of the instruction opcode sequences for the x86/x64 Intel CPUs.

The File | Load File | Flirt Signature menu item provides a list of available compiler and library signatures that can be applied to the disassembly in order to sift through the boilerplate and standard known code and focus on the malware-specific code. FLIRT stands for Fast Library Identification and Recognition Technology, which is how IDA Pro nametags vendor-specific compiler assembly output and libraries and applies the templates as signatures to the loaded disassembly code.
You can choose any one of them at a time and press OK to have it loaded into IDA Pro.

File | Produce File | Create ASM and Create LST are two nice options for taking out paper printouts of the LST listings file and the ASM assembler dump from IDA Pro. The uses are myriad, from automation building to manual note taking. If you have ever had the privilege to work with earlier disassemblers such as W32Dasm, you will feel right at home with this text dump-based format.
Knowing your bearings in IDA Pro

Navigation is quite intuitive and mainly done using double-clicks and scrollbars using the left-mouse button or the mouse middle scroll wheel. Going back to the previously visited addresses requires pressing the Esc key. Links (subroutines and memory offsets such as the jxx/call destinations and the loc_xxxx destination labels) and Code XREF or Data XREF (also known as strings) (cross references for transporting to the cross-referencing item in the display) are the primary ways to navigate through code in IDA Pro by double-clicking on them.

You can navigate through the history by using the backward and forward buttons and view the available items in the buffer via drop-down arrows. Alternatively, if you want to go to a specific address, you can press G and type the virtual address or a named location in the box.

The navigation band is unique to IDA Pro as it is the only disassembler to implement this particular navigation control.

The yellow bar hanging from the top represents the current location in the IDA view. The teal-colored bands represent the FLIRT-recognized library code. Light pink denotes the imports section. Gray and brown indicate the defined and undefined data. Dark blue represents the user code.
Pressing F12 (Flow Chart) and Ctrl + F12 (Function Calls) produces graphs that give an overview of the call sequences via cross references and possible pathways.

From the Graph menu or the right-click context menu in a function in the disassembly, you get the Xrefs from menu item, which analyzes all cross references (function calls and library calls) branching out from the current function.
Hooking up with IDA Pro

The following image shows the IDA Pro Plugins group under Edit | Plugins:

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Keyboard Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 Emulator</td>
<td>Alt+F8</td>
</tr>
<tr>
<td>Universal PE unpacker</td>
<td></td>
</tr>
<tr>
<td>Sample plugin</td>
<td>Alt+0</td>
</tr>
<tr>
<td>Jump to next fixup</td>
<td>Alt+F12</td>
</tr>
<tr>
<td>Hex-Rays</td>
<td></td>
</tr>
<tr>
<td>COM Helper</td>
<td></td>
</tr>
<tr>
<td>Sample chart builder</td>
<td></td>
</tr>
<tr>
<td>Change the callee address</td>
<td>Alt+F11</td>
</tr>
<tr>
<td>Universal Unpacker Manual Reconstruct</td>
<td></td>
</tr>
<tr>
<td>zynamics BinExport 5</td>
<td></td>
</tr>
<tr>
<td>zynamics BinDiff 4.0</td>
<td>Ctrl+6</td>
</tr>
</tbody>
</table>

Quite a few plugins use a modifier to work within IDA, such as the x86 Emulator plugin (Alt + F8) and zynamics BinDiff (Ctrl + 6).

Hex-Rays is a decompiler that cooks up a C code-like source representation from the disassembly. You need to select the required region and press F5.

To use zynamics BinDiff, you will need to copy the installation plugins to the IDA Pro plugins folder. Thereafter, upon restarting IDA Pro, the plugin appears in the Plugin menu. Pressing Ctrl + 6 brings up a Diff database load dialog box for the secondary database to load in order to compare to the current one already loaded in IDA Pro. You get the statistics and listings for the matched and unmatched functions in new tabs.
Thereafter, to view the flow graph in the zynamics GUI from IDA Pro, press Ctrl + E, which will open the zynamics BinDiff GUI with the flow graphs loaded for a structural and semantic comparison.

In the preceding figure, the Matched Functions tab displays the various post analysis parameters such as the EA primary (Effective Addresses of the first file), EA secondary, similarity, and confidence; these are values that are normalized from 0.00 to 1.00 with higher values that reflect the degree of success of the matches. The other columns inform you of the matching algorithm used and the algorithm statistics such as the number of code instructions and edges in the detection algorithms (Edge flow graph MD Index/Hash matching/Call reference matching, Edge Callgraph MD Index, and Edges Proximity MD Index, among others).

The zynamics BinDiff GUI can be invoked from the IDA plugin interface, which displays a dual pane interface for side-by-side comparisons of the call graphs with a plethora of graph analysis options. It is highly recommended for complex malware analysis, pattern matching, signature creation, and generics analysis.

Chris Eagle’s x86 Emulator is certainly worth having a look at. The Step, Skip, Run, Run To Cursor, and Jump to Cursor buttons and the registers pane have a functionality similar to that of a debugger. Heap memory and stack memory can be emulated, and dumping from an emulated memory is supported, which would be good for manual unpacking. Breakpoints can be added and removed with a real-time display in the IDA Pro view. Function return values can be emulated.
Threads can be switched. The Bochs debugger is a welcome addition to an emulated dynamic analysis, which can be found in the **Debugger** menu.

![x86 Emulator - thread 0x700 (main)](image)

**Entropy**

The byte distribution of any binary file in your computer has certain entropy to it. Entropy can be simply defined as a measure of disorder or uncertainty in a given system.

To explain the value of this metric in more simplistic terms, since file (binary/text) structures follow a set template for the most part, the data structures associated with it develop certain expected patterns. The rules that give the file meaning to its parser or parent software expect grammar rules to be followed. However, if the data structure is random and does not follow a set sequence, the rules that expect the structure in sequence will fail to validate the input stream (series of bytes). This incoherence or discrepancy will be directly proportional to the entropy of the file or the selected regions thereof. This would mean that the file is either filled with junk and random data or a custom format, or the data is corrupted or packed, compressed or encrypted, or any combination thereof. However, as more information can be accumulated with such systems, the sample data can be used to reduce the entropy and deal with failure conditions by an analysis of the input and getting a clearer scope of the sample parameters.
A byte probability distribution is a sum of the probabilities of each byte occurring in the entire file. A byte can have values from 0 to 255 in decimals. Notated in hexadecimals, the values are from 0x00 to 0xFF. The probability of each byte occurring in the file stream is as follows:

\[ P(b) = \frac{\text{total count of individual byte value occurrences in the file}}{\text{total number of bytes in the file}} \]

Taking the sigma (or summation) of each of these probabilities and mapping or normalizing the value to a negative logarithmic scale gives us a value from 0.0 to 8.0 when calibrated to mean the 8 bits used to encode a byte, or the number of bits required to represent a byte in the current data stream.

\[ \text{Entropy} = -\Sigma(0 \text{ to } N \text{ samples})\{P(b) \times \ln (P(b))\} \]

The values can be in fractions as well. The negative of the logarithm is taken to remove the negative sign for base 2 log values of negative powers. \( \ln(1/8) = -3 \) because \( 1/(2^3) = 2^{-3} \). Probabilities will normally be between 0 and 1, unless the data expected has a probability of 1, such as a data input stream where each byte occurs with equal probability. Say for a length of a byte input stream of size 256, where every byte from 0–255 occurs exactly once, you have a per byte equal probability of 1/256.

We know that \( \log_2 (1/256) = \ln(1/256)/\ln(2) = -8 \)

For each byte, the value of the expression \( \{P(b) \times \ln(P(b))\} \) will be \( -(1/256 \times 8) \).

Perform a sigma operation as follows: \(-1 \times 256 \times -(1/256 \times 8) = 8\). Now that we know the significance of the negative sign, we can say that the entropy is 8. Information theory-wise, it would mean that the file has a lot of information. However, for our purposes, this file certainly has no defining structure, other than the fact that the distribution is anomalously uniform and contains all the information that it can have in a file, or all events have occurred that could occur within the range of possible events.

A base 2 logarithm is the number of bits (information units) that are required to represent or distinguish \( n \) number of states/symbols. It boils down to permutation and statistical metrics represented in another more compact manner.

The following is the code in C#, which is a class that gives the entropy value as a string. The class exports a static method, and hence, there is no need to make an instance in an OOP paradigm; further, it can be used in any of the .NET-supported languages.
The method can be called using the following:

```csharp
string value = Entropy.GetEntropy(<byte array of the input file>);
```

You need to pass the byte array of the input file.

In C#, you can use the `File` class and the `ReadAllBytes()` method that returns a byte array object.

```csharp
namespace ENTROPY
{
    class Entropy
    {
        public static string GetEntropy(byte[] c)
        {
            int[] numArray = new int[0x100];
            byte[] buffer = c;
            for (int i = 0; i < 0x100; i++)//initialize each element to zero
            {
                numArray[i] = 0;
            }
            for (int j = 0; j < (buffer.Length - 1); j++) //histogram of each byte
            {
                int index = buffer[j];
                numArray[index]++;
            }
            int length = buffer.Length;
            float entropy = 0f;
            for (int k = 0; k < 0x100; k++)
            {
                if ((numArray[k] != 0) && (k != 0))
                {
                }
            }
            return entropy.ToString();
        }
    }
}
```
Analyzing a sosex_64.zip from the http://www.stevestechspot.com/downloads/sosex_64.zip file will give you a value of 7.96, which is a very high entropy value. You can read more on building a visualizer component in C# for an entropy analysis at http://resources.infosecinstitute.com/building-custom-controls-in-c-part-1/.

Some range normalizing or scaling methods compact the range of values from 0 to 1 and can be used in probability distributions. Taking a reciprocal is one of the most common and simplest methods with the other variants working on the mathematical properties of $e$ to map to sigmoid or hyperbolic curves on a plot:

Sigmoid ($X$) = $1/(1+e^{-X})$

Hyperbolic($X$) = $(e^{2X} -1)/(e^{2X}+1)$

Reciprocal ($X$) = $1/X$

Visit the following links to learn more about them:

- https://en.wikipedia.org/wiki/Hyperbolic_function

For our purposes, the final value represents the number of bits required to get information out of the input stream. If the value is high, the byte stream is most likely encrypted or obfuscated or is simply junk corrupted data, but you still need to differentiate it by using other analyses to complement the initial red flags.

Entropy analysis is a very useful metric to detect compressed files, encrypted files, packed files, and obfuscated data, and hence, is indispensable to malware analysis and malware forensics. Compiled code rarely gives this kind of randomization as it follows strict grammar according to the source code text. Hence, when binary executables are tampered with or armored in any way, this simple metric can give away that fact. You can think of entropy as an anomaly detector for a given rule set for our purpose of malware analysis.
Summary

In this rather quick tour, you learned about number systems in depth and looked at how binary, hexadecimal, and decimal notation schemes work. You have also got a clear idea of how negative number representation methods and 1's complement and 2's complement representations work in computing. You examined what logic gates are and how bit masking works.

You looked at the tool chain and some of the most useful tools that will immensely aid you in your static analysis tasks. You had a better look at PE Explorer and IDA Pro, as well as discussed the myriad ways in which the tools can be used. In the next chapter, we will take a deeper look at some of the important data structures and how to use a debugger and disassembler in tandem to get the best out of your analysis session. As we progress, you will also get to learn about debugger internals, a deeper exploration of malicious code, which will aid you in your antimalware pursuits. See you there!
Where to buy this book


Click here for ordering and shipping details.