MATT Common Test Cheatsheet

# Introduction to Malware Analysis Tools & Techniques

## Definition of Malware

Malware is a software that is created with a malicious intent. It is used to compromise computer functions, steal data, bypass access controls and cause harm to other host computers

## Malware Symptoms

* Increased CPU usage
* Device slowing down
* Issues with internet connection
* Freezing / crashing
* Modified / Deleted Files
* New files appera
* Programs running, switched off or reconfigured
* Strange device behavious
* Emails / message sent without knowledge

## Types of Malware

| Malware | Malware Description |
| --- | --- |
| Adware | Type of malware that automatically delivers advertisement (Pop ups) |
| Bot | Program created to automatically perform specific operations |
| Bug | Flaw in a software that produces undesired outcomes. Security bugs are the most severe bugs and allow attackers to deliever exploits |
| Ransomware | Malware that holds computer system captive while demanding ransom |
| Rootkit | Malware to remotely access and control a device without being detected |
| Spyware | Malware to monitor user activity without their knowledge |
| Trojan Horse | Disguises as a normal file to trich users into installing the malware |
| Virus | Attach to files and programs and await for the execution to infect, Self replicating |
| Worm | Spread over the network |

## Overview of MATT

1. Basic Static Analysis
2. Basic Dynamic Analysis
3. Advanced Static Analysis
4. Advanced Dynamic Analysis

Static is not executing malware  
Dynamic is executing the malware

### Basic Static Analysis

* Examine executable file without viewing actual instructions
* Can confirm if the file is malicious
* Straightforward and quick
* important behavious can be missed from complex malware

### Basic Dynamic Analysis

* Requires execution of malware
* Observing the behaviours
* Require safe environment to run malware

### Advanced Static Analysis

* Reverse engineering
* loads executable in a disassembler
* look at the program instructions to discover functions of program
* Steeper learning curve
* require specialized knowledge on disassmbly, code construct and Windows OS

### Advanced Dynamic Analysis

* Execute Malware in a debugger
* View line by line to see what the code does
* Reverse Engineering
* extract detailed information from an executable
* most useful when attempting to gather information with other techniques

# Analysing Windows Malware

## What is Windows Malware?

* Most popular operating system
* Malware that interacts with the operating system
  + Uses Application Programing Interface (API) to execute

## Windows API

### Hungarian Notation

* Windows uses own naming for the different data types
* Identifier naming convention in computer programming
* Used for API function identifiers
* Use a prefix naming scheme to easily identify the type of variable

| Type and Prefix | Description | Size in Bytes |
| --- | --- | --- |
| Word (w) | 16-bit unsigned value | 2 bytes |
| DWORD (dw) | A double-WORD, 32 bit unsigned value | 4 bytes |
| Handles (H) | Reference to an object that is not documented and should only be manipulated by the Windows API | - |
| Long Pointer (LP) | A pointer to a data type. E.g. LPCSTR - Long Pointer Character String | - |
| Callback | Function called by Windows API | - |

### What is a Handle

* Items opened by thge Operating system.
* Point to a location in memory
* Handles can be stored for a later use
* Arithmatic operations cannot be performed

### File System Functions

Create File

Create or open files  
Copy

ReadFile / WriteFile

- Used to read / write to files  
- Operates on file as a stream  
Copy

CreateFileMapping and MapViewOfFile

- Allows manipulation of files in the loaded memory  
- Provides easy manipulation therefore commonly used in malware  
- CreateFile Mapping loads file from disk to memory  
- MapViewOfFile returns pointer to base address of the mapping used to access file  
- The pointer at the base address used to read / write can navigate the file easily  
- extremely handy for parsing files  
Copy

## Windows Registry

* Used to store OS and program configurations
* Good source of host-based indicators
* Uses a hierarchial database of information to improve performance
* Most configurations are in registries

### Why Malware uses registry?

* Persistence
* Configuration

### What is in the Registry

| Registry Terms | Purpose |
| --- | --- |
| Root Key | Registry is divided into 5 top-level sections called root key / hives. (HKEY) |
| Sub Key | Similar to sub folder to a folder |
| Key | can contain other folders. E.g. Root Key and Sub Key |
| Value Entry | Ordered Pair with name and value |
| Value or Data | Stored in the registry entry |

#### Root Keys

| Key Name | Stored Settings |
| --- | --- |
| HKEY\_LOCAL\_MACHINE(HKLM) | Global Settings for the machine |
| HKEY\_CURRENT\_USER(HKCU) | Specific settings for current user |
| HKEY\_CLASSES\_ROOT | Settings on Information defining types |
| HKEY\_CURRENT\_CONFIG | Settings on current hardware configurations |
| HKEY\_USERS | Settings for the current user, new users and default users |

### What are the Common Registry Functions

| Function Name | Purpose |
| --- | --- |
| RegOpenKeyEx | Opens registry for querying or editing |
| RegSetValueEX | Add new value to the registry and set the data |
| RegGetValue | Returns the data for a value entry |

### Registry Tools

| Tools | Purpose |
| --- | --- |
| Regedit | To edit registry entries |
| Autoruns | Parse registry to find entries that start the applications on boot up |
| Regshot | Take a snapshot of the current state of the registry |

## Networking API

* Program uses socket to listen and send data to network
* Common API is located at WS2\_32.dll

| Function | Description |
| --- | --- |
| socket | Creates a socket |
| bind | Attach a spclet to a particular port, prior to the acceot call |
| listen | Indicates that a socket will listen for incoming connections |
| accept | Opens a connection to a remote socket and accepts the connection |
| connect | Open a connection to a remote socket and the remote socket must be waiting for connection |
| recv | Recieves data from the remote socket |
| send | Sends data to the remote socket |

### Sniffers

1. Create a RAW Socket using WSASocket() / socket()
2. Bind socket to interface bind()
3. Put the interface into promiscuous mode.

Promiscuous mode

Mode for wired netwoek interface controller that cause the controller to pass all traffic to a central processing unit instead of just to the controller intended to recieve.  
Copy

## WinINET API

* Higher API implement higher level protocols
* InternetOpen
* Initialise a connection to the internet  
  Copy
* InternetOpenURL
* Connect to a URL  
  Copy
* URLDownloadToFileA
* Download a file from the internet  
  Copy
* InternetReadFile
* Read a file off the internet  
  Copy

### Downloaders

1. URLDownloadToFile() to download a file off the internet
2. ShellExecute() / WinExec() to exexute the newly downloaded file

## Process Manipulation

* Malware creates new processes to evade security within the system
* Win32 API is used - CreateProcess
  + used to create new processes
* The parameter STARTUPINFO includes a handle to standard input, output and error messages
* Malicious Programs can use the process to connect to sockets allowing remote shell execution

## Keyloggers

* Monitor users keystrokes
* Used to spy on users to collect information

### Common methods for Keyloggers

* Install a hook for keyboard events
* Poll Keyboard state with GetAsyncKeyState()

#### Hooking the keyboard

Hook - Mechanism to intercept actions made by user.

Hook Procedure - Function that intercepts certain types of event

Act on events and modify / discard the event

* API: SetWindowsHookExA
  + Called with WH\_KEYBOARD parameter
  + event is relayed to the malicious function when the kwy is pressed
* API: SetWindowsHookExA
  + Called with WH\_MOUSE parameter
  + Intercepts mouse click messages

### Polling the Keyboard

* Malware goes into a loop
* Malware poll the state of every key
* GetAsyncKeyState is called to get hte state of a specific key
* Parameter is the key being pressed

# Basic Static Analysis

## Malware Fingerprint

1. Scan the file with an Antivirus scanner (VirusTotal)

* AntiVirus Scanner uses
  + File Signatures
    - Identifiable pieces of suspicious code
  + Heuristics
    - Behavious and pattern-matching analysis
* Flaws in Antivirus
  + Malicious file identified by an Antivirus may not be identified by another Antivirus
  + Scans suspicious files using different Antivirus engines
  + Generates report and provide total number of Antivirus that flags the suspicious files
  + Acquired by google

1. Hashing (WinMD5)
   * Common way to fingerprint malware
   * Generated by a hashing alogorithm
   * One way function
     + cannot be recovered

## Finding Strings in Malware

1. Strings (BinText)
   * Sequence of characters terminated with a NULL (0x00) character
   * can reveal information on the program
     + URLs
     + Filenames
     + Windows Function Calls

* Types of Strings
  + ASCII
    - 1 byte per character
    - represented in hexadecimal format
  + Unicode
    - 2 bytes per character

## Portable Executable (PE) (Dependency Walker)

* PE File Format used by Windows executables, object code, DLLs
* PE File Format is a data structure that contains necessary information for Windows OS Loader to execute compiled codes
* Begin with header that includes information like
  + Coder
  + Type of application
  + Required libraries
  + Space requirements

Programmer writes a source file. The source file gets translated into an object module. The linker than combines the various object modules into an executable image.

Once the executable image is loaded, it will execute the image on the system.

An executable file is a complete program that can be run directly by an operating system (in conjunction with shared libraries and system calls). The file generally contains a table of contents, a number of code blocks and data blocks, ancillary data such as the memory addresses at which different blocks should be loaded, which shared libraries are needed, the entry point address, and sometimes a symbol table for debugging. An operating system can run an executable file more or less by loading blocks of code and data into memory at the indicated addresses and jumping to it.  
  
Most programs are written with source code logically divided into multiple source files. Each source file is compiled independently into a corresponding "object" file of partially-formed machine code known as object code. At a later time these "object files" are "linked" together to form an executable file.  
  
Object files have a lot in common with executable files (table of contents, blocks of machine instructions and data, and debugging information). However, the code isn't ready to run. It is full of incomplete references to subroutines outside itself, and as such, many of the machine instructions have only placeholder addresses.  
The linker, as a final phase of compilation, will read all of the object files, resolve references between them, perform the final code layout in memory that determines the addresses for all the blocks of code and data, fix up all the placeholder addresses with real addresses, and write out the executable file.  
Copy

### Loading a Portable Executable (PE)

#### Types of loading

1. Absolute Loading  
   - Load program at same address everytime
2. Relocatable Loading  
   - Load program at different address based on what is available
3. Dynamic Run-time loading  
   - Load and reload program at different address while program is running.

#### Address Binding

* Must be brought to main memory before execution
* instructions that use addresses must be properly bounded to address space in memory
* Scheme that perform the job
* Symbolic name / label bounded to the actual address
* actual binding can be specified / resolved at
  + Compile time
  + Link time
  + Load time
  + Run time

A program, to be executed, must be brought to main memory. The instructions that use addresses in a program, must be bound to proper address space in main memory. Address binding is a scheme that performs this job. It can be thought as a mapping from one address space to another. Symbolic label/name is translated (bound) to an actual address.

#### Types of linking

1. Static Linking
   * Statistically linked to a binary
   * All executable code from library is copied into the executable
   * Difficult to differentiate the code between statistically linked and executables code
   * PE header does not contain such information
2. Dynamic Linking
   * Executable connect to library only when function is needed
   * Executable search and load correct library at runtime
   * address of library function to connect resolved at runtime
   * Information to resolve address stored in .idata (import section)

#### PE File Format

#### Relative Virtual Address

* Virtual Address that is relative to the base address of the executable program
* A base address is a unique location in primary storage (or main memory) that serves as a reference point for other memory locations called absolute addresses.
* In order to obtain an absolute address, a specific displacement (or offset) value is added to the base address. In primary storage, all addresses literally comprise fixed-length sequences of bits that stand for positive whole numbers usually expressed in hexadecimal form. For example, a base address might indicate the beginning of a program loaded into primary storage. The absolute address of each individual program instruction could be specified by adding a displacement to the base address.
* Address may not be known at link time
* Addresses stored in code by linker must be relative

#### Base Address, Relative Address

Base Address that serves as a reference point for other address.  
Relative Address of an instruction is the offset from the Base Address

### PE File Format

* Starts with DOS Header
  + Signature "MZ" (4D 5A, Mark Zbikowski - Developer of MS-DOS)
  + Offset to PE Header
* PE Header
  + Machine Type
  + Number of Sections
  + Timestamp
  + Data Directory (Where the data is stored)
* Section Table
  + List of Sections
* Sections

### PE Sections

| Field | Purpose |
| --- | --- |
| .text | Executable Code |
| .data | Read / Write initialised data |
| .rdata | Read only data |

* Note that the linker combines text and data from various object modules to form the executable image.
* Compilers can append “$…” to the end of the names to dictate the ordering within a section.
* For example “.text$X"isbefore``.text$Y” in the .text section

#### .edata (Exports)

* Run image required by calling the dll
* Loader must be able to find entry points into the dll
* Export table stores the list of addresses that the dll can call
* Exported entry point has a unique ordinal value
* entry point requires to know dll name and ordinal number
* Programmers are unsure of ordinal number
* export table used to translate names to ordinal numbers

#### Base Relocation

* addresses must be fixed when module is relocated
* .reloc specifies location of where the module is moved to

#### .resrc

* Resources for images
* Examples: Bitmaps and icons
* Organised like file systems

#### .debug

* Debug information
* Was “coff” up to NT 4.0 and has moved onto “pdb” in Window XP

## Packed Executables

* Obfuscated to make it hard to analyse
* Aim to hide malware functionality
* Obfuscation through
  + Compression
  + Encryption
* EXE is not readable when encrypted

Small wrapper stub to decompress / decrypt original before loading to memory  
Only stub is analysable when program is packed  
Copy

### Detecting Packers

* PEiD
* Has inbuild packers like
  + UPX
  + ASPack
  + PELock
  + Themida

# Basic Dynamic Analysis

## Why Dynamic Analysis?

* Performed by executing the malware
* performed when static analysis reaches dead end
* Allow the observation of malware's true functionalities
* Places system at risk

## Sandbox

* Analyse malware automatically and produce a report
* Provide easy-to-understand output
* Good for initial triage (use for fast analysis of malware - automatic)
* All sandboxes are similar in approach

### GFI Sandbox Overview

1. Malware is submitted to repository  
   - User  
   - Scripted submission
2. Malware is inserted into repository
3. Samples are retrieved from sandbox based on queue and priority
4. Analysis is sent back to repository
5. User view analysis result via web interface

#### GFI Sandbox Report Sample

* Basic Static Analysis
* High level dynamic analysis
* File activity Provided
  + Created
  + Read
  + Opened
  + Deleted
* Mutex created
* Changes to registry
* Network Activity by malware

#### What is a mutex

- Program or object that allows multiple program thread to share the same resources, but not simultaneously  
- Uniquely named on program start  
Copy

### Sandbox drawbacks

* Command line options
* Waiting for some command-and-control instructions
* Not all events may be recorded because the Sandbox did not wait long enough
* Malware may detect that it is running in a Sandbox
* Conditions are not met for Malware to run properly
* Sandbox OS may not be correct for the malware to run properly.

## Running of Malware

Malware comes in 2 forms

1. Executable (EXE)
2. Data Link Library (DLL)

* Installing a dll  
   rundll32.exw DLLname, Export arguments  
    
  DLL often runs most of their codes in their DLLMain Function  
  If the DLL needs to install a service ;   
   rundll32 ipr32x.dll, installService ServiceName  
   net start service name  
  Copy

## Process Monitor

* Combines functionalities of filemon and regmon
* Monitors
  + Registry Activity
  + File Activity
  + Process Activity
  + Network Activity
* Monitor system call for all processes by default
* Need to filter events (Has ready made filters)

## Process Explorer

* Provide insights to running processors
* Features
  + List active processes
  + DLL loaded for each process
  + Process properties
  + Active TCP connections
  + Create / Kill / Validate Processes

### Verify Option

* Verify if processed has been digitally signed by Microsoft
* Only apply to process on disk

## Monitoring Registry (RegShot)

* Regshot is opensource registry comparison tool
* Compares two registry snapshots and report differences

## CaptureBAT

* Behaviour Analysis Tool for WIN32 Applications
* Monitor state of system during execution of applications and processing of documents
* Performed at low kernal level
* Very good to exclude event noise that naturally occur in the system

## Monitoring Network Activity

* Malware send information to another server
* Tries to connect to a command-and-control server to await instructions
* requires a faking of network
* Able to reveal most functionalities

### ApateDNS

* Free tool from Mandiant
* Able to
  + List DNS request made by malware
  + Spoof DNS responses
  + Set NXDomain (Error message if domain doesn’t exist)

### Netcat

* TCP-IP Swiss Army Knife
* Used for inbound / outbound connections
  + Port Scanning
  + Tunneling
  + Proxying
  + Port forwarding
  + Etc.

### Wireshark

* Opensource network sniffer
* Captures network packet
* Provides
  + Visualisation of network packets
  + Packet-stream analysis
  + In-depth packet analysis

# Assembly Part 1

## Why Assembly Language?

* Malware can be found in portable executable files
* Basic static and dynamic malware analysis methods do not provide enough information to analyze malware completely
* Analysing malware codes in Assembly language provides more complete information

## What is Assembly Language?

* Machine Language is interpreted on a computer architecture
* Representation of machine language
* Obtained from disassembly of binary code

## What are levels of abstration

* levels of abstraction that create a way of hiding the implementation details
* levels start from the bottom
* Higher level of abstractio placed near the top with more specific concepts.
* Lower levels result in less portable the level accross computer systems

## x86 Architecture

Van Neumann Architecture Consists of 3 hardware components

* Central Processing Unit (CPU) executes code
* Main memory stores all data and code
* Input / Output system (I/O) interface with devices such as hard drives, keyboards and monitors

1. CPU gets instructions to execute from RAM using a register (Instruction Pointer)

* Instruction Pointer stores address of the next instruction to execute  
  Copy

1. Register stores the address of the instructions to execute

* Registers are the CPU’s basic data storage units and are often used to save time so that the CPU doesn’t need to access RAM.  
  Copy

1. Arithmatic Logic Unit (ALU) executes the instructions fetched from the RAM and places result in registers / memory
2. Process of fetching and executing instruction after instruction is repeated when the program runs

### Main Memory

* Can be divided into 4 major sections
* Order of the sections may not be the same

1. Data Section
   * Refer to specific section of memory
   * Contain values put in place when the program is initially loaded
   * Sometimes called Static values - Do not change when program is running / Global values
2. Code
   * Includes instructions fetched by CPU to execute program task
   * Control what the program does and how the program tasks are orchestrated
3. Heap
   * Used for dynamic memory during program execution
     + create (allocate) values
     + eliminate (free) values
   * refered as dynamic memory
     + contents can change frequently when program is running
4. Stack
   * Used for local variables and parameters for functions
   * Help control Program flow

### Instructions

#### Instruction Format

mnemonic and zero or more operands

MNEMONIC word that identifies the instructions to execute  
Example: mov  
  
OPERANDS used to identify information used by the instruction  
Example: registers or data  
Copy

#### Opcodes and Endianness

* Each instruction correspond to opcodes (Operation Codes)
* Tell CPU the operation the program wants to perform
* Disassembler tranlate opcodes into human-readable instruction

0x42000000 is treated as 0x42 as the architecture uses little endian format

Endianness of data describe wheter the most significant bit (big-endian) or least significant bit (little-endian) byte is ordered first (at the smallest address) within a larger data item

Require conversion when malware makes use of network communications

#### Operands

* Used to identify data used by the instruction
* Has 3 ways it can be used in
  1. Immediate operands are fixed value
  2. Register operands refer to registers like ECX
  3. Memory Address operands refer to memory address that contains the value of interest
     + Denoted by a value, register, or equation between brackets
     + Such as [eax].
* Opcode is a machine language equivalent of an executed assembly instruction
* Identifiers are not case-sensitive

### Registers

* Small amount of data storage available to CPU
* Content can be accessed more quickly than storage available
* x86 processors have collection of register for use as temporary storage
* 4 register categories
  + General registers are used by CPU during execution
  + Segment registers are used to track sections of memory
  + Status registers are used to make decisions
  + Instruction Pointers are used to keep track of the next instruction to execute

#### Register size

### General Registers

* Typically store data or memory address
* Used interchangably to get things done within the program
* Some instructions use specific registers by defintion
  + Division use EDX
  + Multiplication use EAX
  + EAX can also hold return value for function call
  + ESP, EBP used for function call/return
  + ESI, EDI and ECX are used in repeat instructions
    - ESI and EDI may store memory address
* General registers can be used in a consistant fashion throughout a program
* Use of registers in a consistent fashion across compiled code is known as a convention
* Allows the quicker examination of code (from convention)

### Flags

* EFLAGS Register is a status register
* 32 bits in size, each bit is a flag
* Flags are set to 1 or 0
* Control CPU operations / Indicate results of CPU operation

#### Important Flags

| Flag | Name of Flag | Description |
| --- | --- | --- |
| ZF | Zero Flag | Set when result of operation is equal to zero |
| CF | Carry Flag | Set when result of an operation is too large or too small for destimation operand |
| SF | Sign Flag | Set when result of operation is negative, cleared when result is positive / set when the most significant bit is set after an arithmetic operation. |
| TF | Trap Flag | Used for debugging. Execute one instruction at a time when flag is set |
| OF | Overflow Flag | Set when an instruction generates an invalid signed result |

### EIP (Instruction Pointer / Program Counter)

* Register that contains the memory address of the next instruction to be executed for a program
* Purpose is to tell the processor what to do next
* When EIP is corrupted, program will likely crash

### Data Allocation

| Syntax | Name | Size |
| --- | --- | --- |
| DB | Define Byte | 1 byte |
| DW | Define Word | 2 byte |
| DD | Define DoubleWord | 4 byte |
| DQ | Define QuadWord | 8 byte |
| DT | Define Ten Byte | 10 byte |
| ? | Uninitialized data | - |
| ; | Comment | - |
| '' | Strings | multiple bytes |

* Strings allocate multiple bytes
* Labels in front of the directives remember offsets from the beginning of the segment accomodating the directive
* Multiple definitions can be abbreviated
* Multiple definition can be initialised in data structures like arrays

### Data Allocation Directives

| Syntax | Purpose | Example |
| --- | --- | --- |
| DUP | Used to initialise duplicated value | DB 10 DUP(?) |
| EQU | Assign result of expression to name, expression is evaluated at assembly time | - |

### C Data Types

| Directive | C data type |
| --- | --- |
| DB | char |
| DW | int, unsigned int |
| DD | float, long |
| DQ | double |
| DT | internal intermediate float value |

### Program Layout

### Assembly Instructions

* mov (move)
* lea (load effective address)
* [] (memory address)
* sub (subtract)
* add (add)
* inc (increment)
* dec (decrement)
* mul (multiply)
* div (divide)

| Instruction | Description |
| --- | --- |
| mov eax, ebx | Copies te contents of EBX into the EAX register |
| mov eax, 0x42 | Copies the value of 0x42 into the EAX register |
| mov eax, [0x4037C4] | Copies the 4 bytes of te memory location 0x4037C4 into the EAX register |
| mov eax, [ebx] | Copies the 4 bytes at the memory location specified by the EBX register into the EAX register |
| mov eax, [ebx+esi\*4] | Copies the 4 bytes at the memory location specified by the result of the equation ebx+esi\*4 into the EAX register |
| movsx eax, bh | (move with sign extend) move the higher byte of the 16-bit register BX sign-extended into the 32-bit register EAX |
| lea ax, [bx] | Puts value in bx into ax |
| lea bx, [bx+3] | Increases value in bx by 3 |
| lea eax, [ebx+8] | Puts value of (ebx + 8) into eax |
| lea ecx, [0+4\*eax+eax] | Multiplies value in eax by 5 and puts it into ecx |
| lea esi, [ebx+8\*eax+4] | Puts value of (ebx + 8\*eax + 4) into esi |
| sub eax, 0x10 | Subtracts 0x10 from EAX |
| add eax, ebx | Adds EBX to EAX and stores the result in EAX |
| inc edx | Increment EDX by 1 |
| dec ecx | Increment ECX by 1 |
| mul value | multiply eax by a value and the result is stored in edx:eax |
| div value | divide edx:eax by value and the result in eax and remainder in edx |

#### Logical Operators

* OR
* AND
* XOR
  + Used to set the register to zero
  + XOR EAX, EAX
* SHR
  + Shift right the registers by the count
  + SHR destination, count
  + CF flag contains the last bit shifted out of the destination
* SHL  
  - Shift left the registers by the count  
  - SHL destination, count  
  - CF flag contains the last bit shifted out of the destination  
  operand
* ROR
  + Shift right the registers by the count
  + bits shifted move to the front of the register
* ROL
  + Shift left the registers by the count
  + bits shifted move to the end of the register

| Instruction | Description |
| --- | --- |
| xor eax, eax | Clears the EAX register |
| or eax, 0x7575 | Performs the logical or operation on EAX with 0x7575 |
| mov eax, 0xA | Shifts the EAX register to the left 2 bits; result in EAX = 0x28, because 1010 (0xA in binary) shifted 2 bits left is 101000 (0x28) |
| shl eax, 2 | Shifts the EAX register to the left 2 bits; result in EAX = 0x28, because 1010 (0xA in binary) shifted 2 bits left is 101000 (0x28) |
| mov bl, 0xA | Rotates the BL register to the right 2 bits; result in BL = 10000010, because 1010 rotated 2 bits right is 10000010 |
| ror bl, 2 | Rotates the BL register to the right 2 bits; result in BL = 10000010, because 1010 rotated 2 bits right is 10000010 |

### NOP

* No Operation
* Proceed to the next instruction
* Used for buffer overflow attacks

### INT

* Interrupt
* INT 21H calls DOS Interrupt Service Routine

# Assembly Part 2

## Conditionals

* instructions that perform comparison
* comparison results alter status flags
* Decision based on value of status flag
* Specific bits (flags) in the status register (EFLAGS) can be set or cleared
* Program executio depends on comparison results
* conditional instructions
  + TEST
    - Identical to the AND
    - Does not modify the instructions
    - Set flags (usually zero flag)
  + CMP
    - Identical to the SUB
    - Operands are not affected
    - Set flags (usually zero flag and carry flag)

### AND

Perform boolean AND operation between each pair of matching bits in two operands

AND Destination, Source  
Copy

### OR

Perform boolean OR operation between each pair of matching bits in two operands

OR Destination, Source  
Copy

### XOR

Perform boolean exclusive-OR operation between each pair of matching bits in two operands

Useful in inverting bits in an operand - Clear the value of the register

XOR Destination, Source  
Copy

### NOT

Perform boolean NOT operation on a single destination operand

NOT Destination  
Copy

### TEST

* Performs a nondestructive AND operation between each pair of matching bits in two operands
* No operands are modified, but the Zero flag is affected

### CMP

* Compares the destination operand to the source operand
  + Nondestructive subtraction of source from destination (destination operand is not changed)
* CMP Destination, Source

#### Comparison with Signed integers

## Conditional Jumps

* Jumps based on
  + Specific Flags
  + Equality
  + Unsigned Comparisons
  + Signed Comparisons
* Conditional jump instruction branches to a label when specific register or flag conditions are met

### Jump based on Specific Flags

| Mnemonic | Description | Flags |
| --- | --- | --- |
| JZ | Jump if zero | ZF=1 |
| JNZ | Jump if not zero | ZF=0 |
| JC | Jump if carry | CF=1 |
| JNC | Jump if not carry | CF=0 |
| JO | Jump if overflow | OF=1 |
| JNO | Jump if not overflow | OF=0 |
| JS | Jump if signed | SF=1 |
| JNS | Jump if not signed | SF=0 |
| JP | Jump if parity (even) | PF=1 |
| JNP | Jump if not parity (odd) | PF=0 |

### Jump based on Equality

| Mnemonic | Description |
| --- | --- |
| JE | Jump if equal |
| JNE | Jump if not equal |
| JCXZ | Jump if CX=0 |
| JECXZ | Jump if ECX=0 |

### Jump based on Unsigned Comparisons

| Mnemonic | Description |
| --- | --- |
| JA | Jump if above (if leftOp > rightOp) |
| JNBE | Jump if not below or equal (same as JA) |
| JAE | Jump if above or equal (if leftOp >= rightOp) |
| JNB | Jump if not below (same as JAE) |
| JB | Jump if below (if leftOp < rightOp) |
| JNAE | Jump if not above or equal (same as JB) |
| JBE | Jump if below or equal (if leftOp <= rightOp) |
| JNA | Jump if not above (same as JBE) |

### Jump based on Signed Comparisons

| Mnemonic | Description |
| --- | --- |
| JG | Jump if greater (if leftOp > rightOp) |
| JNLE | Jump if not less than or equal (same as JG) |
| JGE | Jump if greater than or equal (if leftOp >= rightOp) |
| JNL | Jump if not less (same as JGE) |
| JL | Jump if less (if leftOp < rightOp) |
| JNGE | Jump if not greater than or equal (same as JL) |
| JLE | Jump if less than or equal (if leftOp <= rightOp) |
| JNG | Jump if not greater (same as JLE) |

## Repeat instructions

* Used for processing multi-byte data (byte arrays)
* Types of Registers
  + ESI register (Source Index Register)
  + EDI register (Destination Index Register)
  + ECX register (Counting Variable register)
* Requires to be properly initialised for instructions to work

## The Stack

* Stores memory for functions, local variables and flow control
* push items onto stack
* pop items off stack
* Has LIFO (Last in First out) structure
* EBP is the base pointer
  + Stays consistent within a given function
  + program can use it as a placeholder
  + keep track of the location of local variables and parameters
* Stack instructions
  + Push
  + Pop
  + Call
  + Leave
  + Enter
  + Ret
* Allocated in a top down format in memory
* Highest Address allocated and used first
* Short term storage only
* frequently stores local  
  variables, parameters, and the return address
* primary usage is for the management of data exchanged between function calls
* Implementation varies between compilers
* Conventional for local variables and parameters to be reference relative to EBP

## Function

* Block of code within a program
* Made to perform a specific task
* Program calls a function and transfer execution flow control to it
* Control is returned to next program instruction after call when function completes execution

### Organisation of Function

Contains

* Prologue
  + Prepare stack and registers for use within the function
* Body
  + Carry out the main task of the function
* Epilogue
  + Restores states of stack and registers prior to call
* Prologue and Epilogue can contain buffer overflow protection code

### Function Calls

* Portions of code within a program that perform a specific task and that are relatively independent of the remaining code  
  -Items associated with each called function are stored in a stack frame
* ESP (Stack Pointer) points to top item
* EBP (Base Pointer) points to start of frame

1. Arguments are placed on the stack using push instructions.
2. A function is called using call memory\_location. This causes the current instruction address (that is, the contents of the EIP register) to be pushed onto the stack. This address will be used to return to the main code when the function is finished. When the function begins, EIP is set to memory\_location (the start of the function).
3. Through the use of a function prologue, space is allocated on the stack for local variables and EBP (the base pointer) is pushed onto the stack. This is done to save EBP for the calling function.
4. The function performs its work.
5. Through the use of a function epilogue, the stack is restored. ESP is adjusted to free the local variables, and EBP is restored so that the calling function can address its variables properly. The leave instruction can be used as an epilogue because it sets ESP to equal EBP and pops EBP off  
   the stack.
6. The function returns by calling the ret instruction. This pops the return address off the stack and into EIP, so that the program will continue executing from where the original call was made.
7. The stack is adjusted to remove the arguments that were sent, unless they’ll be used again later.

### Push Operation

* A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.
* The Stack grows downwards
* Area below ESP is always available (Unless stack has overflowed)
* PUSH syntax:
  + PUSH r/m16
  + PUSH r/m32
  + PUSH imm32

### Pop Operation

* Copies value at stack[ESP] into a register or variable.
* Adds n to ESP, where n is either 2 or 4
  + value of n depends on the attribute of the operand receiving the data
* POP syntax:
  + POP r/m16
  + POP r/m32

## Basic Constructs

* Often written in high-level language like C
* Main Method of C Program
* Parameters argc and argv are determined at runtime
* argc
  + integer that contains number of arguments on the command line, including the program name
* argv
  + pointer to an array of strings that contain the command line arguments
* .text is the PE file section where the compiler puts the code
* The first line is a comment, indicated by ;
* \_main is a label, used to reference the address of the main function, 00401150
* near indicates that the referenced address is in the same section (.text)
* endp - pointer to string pointer array passed to main (in C++)

## Recognising If Else

## Recognising Loops

# Advanced Static Analysis

## Recognising C Constructs

* Malware commonly developed using C
* Recognise assembly instructions as group on addition to single instructions
* Malware analyst’s goal is to understand overall functionality of program
* Focus on the way a program works in general

#### Global VS Local

### Arithmetic Operations

### If Constructs

### For Loops

### While Loops

### Function Calls

### Switch Construct

### Struct Construct

A struct in the C programming language (and many derivatives) is a complex data type declaration that defines a physically grouped list of variables to be placed under one name in a block of memory, allowing the different variables to be accessed via a single pointer, or the struct declared name which returns the same address.

# Calculating Parameters of a function

Count the number of push before the function call  
Copy

# IDA Interface

Local Variables

Pointers are negative  
Copy

Global Variables

Pointers are positive  
Copy