Institute of Forensic Science
Gujarat Forensic Sciences University

M. Sc. DFIS – II

Unit – 3 [Debugging and Reverse Engineering]

Parag H. Rughani, Ph. D.

Table of Contents

Intro to x86 Assembly

- *Refer: Assembly1.pdf (Discussed and Circulated earlier)*

IDA Pro

- The Interactive Disassembler Professional (IDA Pro) is an extremely powerful disassembler distributed by Hex-Rays.
- Although IDA Pro is not the only disassembler, it is the disassembler of choice for many malware analysts, reverse engineers, and vulnerability analysts.
- IDA Pro also supports several file formats, such as Portable Executable (PE), Common Object File Format (COFF), Executable and Linking Format (ELF), and a.out.

IDA Pro

- IDA Pro disassembles an entire program and performs tasks such as function discovery, stack analysis, local variable identification, and much more.
- IDA Pro includes extensive code signatures within its Fast Library Identification and Recognition Technology (FLIRT), which allows it to recognize and label a disassembled function, especially library code added by a compiler.

IDA Pro

- IDA Pro is meant to be interactive, and all aspects of its disassembly process can be modified, manipulated, rearranged, or redefined.
- One of the best aspects of IDA Pro is its ability to save your analysis progress: You can add comments, label data, and name functions, and then save your work in an IDA Pro database (known as an idb) to return to later.
- IDA Pro also has robust support for plug-ins, so you can write your own extensions or leverage the work of others.

IDA Pro

- When you load an executable, IDA Pro will try to recognize the file's format and processor architecture.
- When loading a file into IDA Pro (such as a PE file), the program maps the file into memory as if it had been loaded by the operating system loader.
- PE files are compiled to load at a preferred base address in memory, and if the Windows loader can't load it at its preferred address (because the address is already taken), the loader will perform an operation known as rebasing.
- This most often happens with DLLs, since they are often loaded at locations that differ from their preferred address.

IDA Pro

- After you load a program into IDA Pro, you will see the disassembly window, this will be your primary space for manipulating and analyzing binaries, and it’s where the assembly code resides.
- You can display the disassembly window in one of two **modes**: **graph** and **text**.
- To switch between modes, press the spacebar.

IDA Pro

- In **graph mode**, the color and direction of the arrows help show the program's flow during analysis.
- The arrow's color tells you whether the path is based on a particular decision having been made: **red** if a conditional jump is not taken, **green** if the jump is taken, and **blue** for an unconditional jump.
- The arrow direction shows the program's flow; upward arrows typically denote a loop situation.
- Highlighting text in graph mode highlights every instance of that text in the disassembly window.

IDA Pro

- The **text mode** of the disassembly window is a more traditional view, and you must use it to view data regions of a binary.
- The left portion of the text-mode display is known as the arrows window and shows the program’s nonlinear flow.
- Solid lines mark unconditional jumps, and dashed lines mark conditional jumps.
- Arrows facing up indicate a loop.

IDA Pro

- **Functions window:**
  - Lists all functions in the executable and shows the length of each.
  - You can sort by function length and filter for large, complicated functions that are likely to be interesting, while excluding tiny functions in the process.
  - This window also associates flags with each function [R (Returns to caller), F(Far), L(Lib), S(static), and so on], the most useful of which, L, indicates library functions.
  - The L flag can save you time during analysis, because you can identify and skip these compiler-generated functions.

IDA Pro

- **Names window:**
  - Lists every address with a name, including functions, named code, named data, and strings.
- **Strings window:**
  - Shows all strings. By default, this list shows only ASCII strings longer than five characters. You can change this by right-clicking in the Strings window and selecting Setup.
- **Imports window:**
  - Lists all imports for a file.

IDA Pro

- **Exports window:**
  - Lists all the exported functions for a file. This window is useful when you're analyzing DLLs.
- **Structures window:**
  - Lists the layout of all active data structures. The window also provides you the ability to create your own data structures for use as memory layout templates.

IDA Pro

- These windows also offer a **cross-reference** feature that is particularly useful in locating interesting code.
- For example, to find all code locations that call an imported function, you could use the import window, double-click the imported function of interest, and then use the cross-reference feature to locate the import call in the code listing.

IDA Pro

- The IDA Pro interface is so rich that, after pressing a few keys or clicking something, you may find it impossible to navigate.
- To return to the default view, choose Windows ▶ **Reset Desktop**. Choosing this option won't undo any labeling or disassembly you've done; it will simply restore any windows and GUI elements to their defaults.
- By the same token, if you've modified the window and you like what you see, you can save the new view by selecting Windows ▶ **Save desktop**.

IDA Pro

- The horizontal color band at the base of the toolbar is the **navigation band**, which presents a color-coded linear view of the loaded binary's address space.
- The colors offer insight into the file contents at that location in the file as follows:
  - **Light blue** is library code as recognized by FLIRT.
  - **Red** is compiler-generated code.
  - **Dark blue** is user-written code.
- You should perform malware analysis in the dark-blue region. If you start getting lost in messy code, the navigational band can help you get back on track. IDA Pro's default colors for data are pink for imports, gray for defined data, and brown for undefined data.

IDA Pro

- To **jump** to any virtual memory address, simply press the G key on your keyboard while in the disassembly window.
- A dialog box appears, asking for a virtual memory address or named location.
- To jump to a raw file offset, choose Jump ▶ Jump to File Offset.
- For example, if you're viewing a PE file in a hex editor and you see something interesting, such as a string or shellcode, you can use this feature to get to that raw offset, because when the file is loaded into IDA Pro, it will be mapped as though it had been loaded by the OS loader

IDA Pro

- Selecting **Search** from the top menu will display many options for moving the cursor in the disassembly window:
  - Choose Search ▶ Next Code to move the cursor to the next location containing an instruction you specify.
  - Choose Search ▶ Text to search the entire disassembly window for a specific string.
  - Choose Search ▶ Sequence of Bytes to perform a binary search in the hex view window for a certain byte order.
  - This option can be useful when you're searching for specific data or opcode combinations.

IDA Pro

- Practical:
  - Explore IDA pro by disassembling various files.

Analyzing Malicious Windows Programs

- Most malware targets Windows platforms and interacts closely with the OS.
- A solid understanding of basic Windows coding concepts will allow you to identify host-based indicators of malware, follow malware as it uses the OS to execute code without a jump or call instruction, and determine the malware’s purpose.
- Non-malicious programs are generally well formed by compilers and follow Microsoft guidelines, but malware is typically poorly formed and tends to perform unexpected actions.
- Windows is a complex OS, so we will focus on the functionality most relevant to malware analysis.

Analyzing Malicious Windows Programs

- The Windows API is a broad set of functionality that governs the way that malware interacts with the Microsoft libraries.
- The Windows API is so extensive that developers of Windows-only applications have little need for third-party libraries.
- One of the most common ways that malware interacts with the system is by **creating or modifying files.**
- File activity can hint at what the malware does.

Analyzing Malicious Windows Programs

- Microsoft provides several functions for accessing the file system, as follows:
  - *CreateFile*: This function is used to create and open files. It can open existing files, pipes, streams, and I/O devices, and create new files. The parameter dwCreationDisposition controls whether the CreateFile function creates a new file or **opens** an **existing one**.
  - *ReadFile and WriteFile*: These functions are used for reading and writing to files. Both operate on files as a stream. When you first call ReadFile, you read the next several bytes from a file; the next time you call it, you read the next several bytes after that.

Analyzing Malicious Windows Programs

- :
  - *CreateFileMapping and MapViewOfFile*: File mappings are commonly used by malware writers because they allow a file to be loaded into memory and manipulated easily. The CreateFileMapping function **loads a file from disk into memory**. The MapViewOfFile function returns a **pointer to the base address** of the mapping, which can be used to access the file in memory. The program calling these functions can use the pointer returned from MapViewOfFile to read and write anywhere in the file. This feature is extremely handy when parsing a file format, because you can easily jump to different memory addresses.

Analyzing Malicious Windows Programs

- Windows has a number of file types that can be accessed much like regular files, but that are not accessed by their drive letter and folder (like c:\docs).
- Malicious programs often use special files as certain special files can provide greater access to system hardware and internal data.
- Some of them are:
  - *Shared Files*: Shared files are special files with names that start with \\serverName\share or \\?\serverName\share. The \\?\ prefix tells the OS to disable all string parsing, and it allows access to longer filenames.

Analyzing Malicious Windows Programs

- :
  - *Files Accessible via Namespaces:* Namespaces can be thought of as a fixed number of folders, each storing different types of objects. The lowest level namespace is the NT namespace with the prefix \. The NT namespace has access to all devices, and all other namespaces exist within the NT namespace. The Win32 device namespace, with the prefix \\.\, is often used by malware to access physical devices directly, and read and write to them like a file. For example, a program might use the **\\.\PhysicalDisk1** to directly access PhysicalDisk1 while ignoring its file system, thereby allowing it to modify the disk in ways that are not possible through the normal API. Using this method, the malware might be able to **read and write** data to an unallocated sector without **creating or accessing files**, which allows it to avoid detection by antivirus and security programs.

Analyzing Malicious Windows Programs

- :
  - **ADS:** The Alternate Data Streams (ADS) feature allows additional data to be added to an existing file within NTFS, essentially adding one file to another. The extra data does not **show up in a directory listing**, and it is not shown when displaying the contents of the file; it's visible only when you access the stream. ADS data is named according to the convention **normalFile.txt:Stream:$DATA**, which allows a program to read and write to a stream. Malware authors like ADS because it can be used to hide data.

Analyzing Malicious Windows Programs

- The Windows registry is used to store OS and program configuration information, such as settings and options.
- Like the file system, it is a good source of host-based indicators and can reveal useful information about the malware's functionality.
- Malware often uses the registry for persistence or configuration data.
- The malware adds entries into the registry that will allow it to run automatically when the computer boots.
- The registry is so large that there are many ways for malware to use it for persistence.

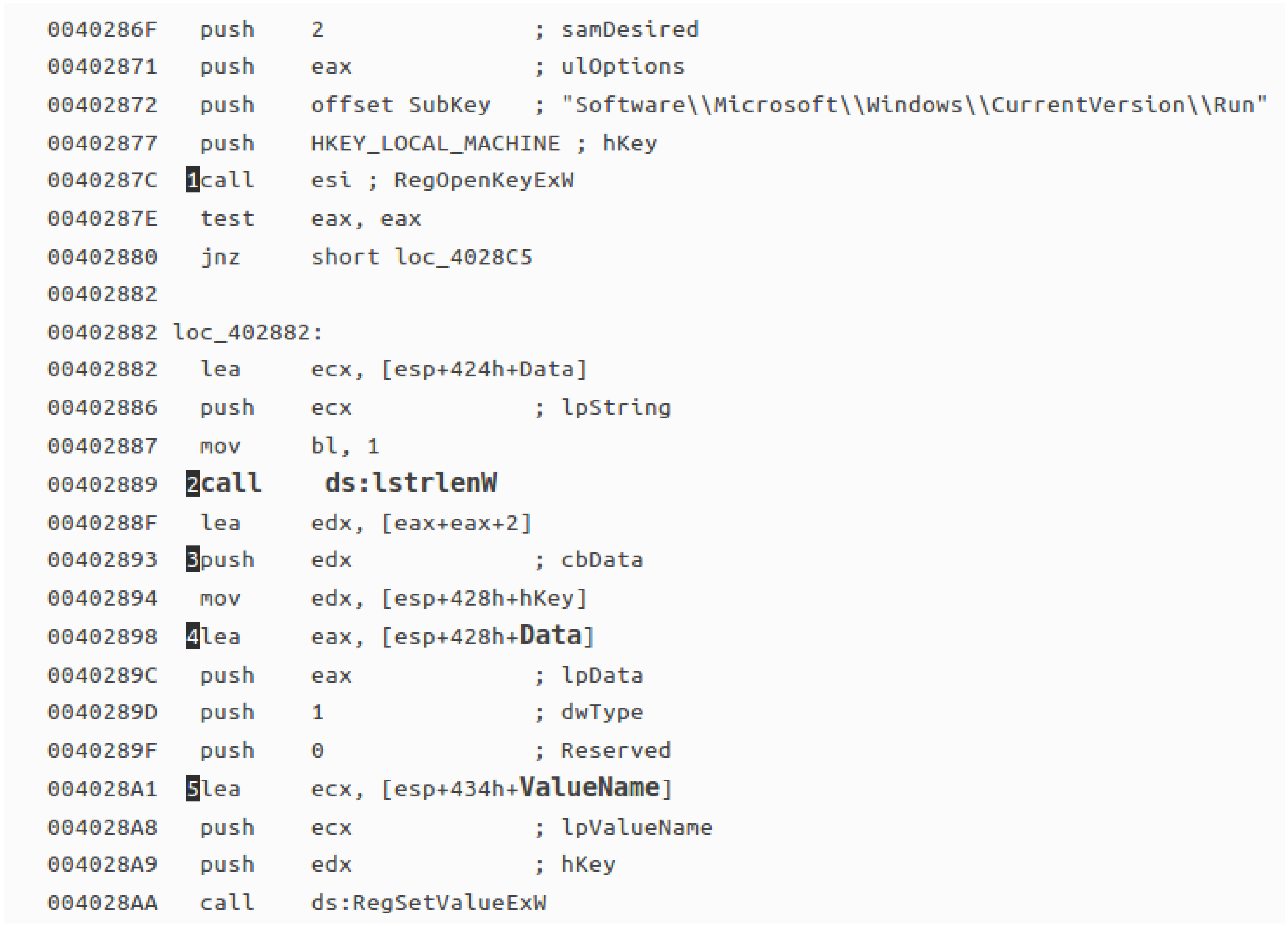
Analyzing Malicious Windows Programs

- Malware often uses registry functions that are part of the Windows API in order to modify the registry to run automatically when the system boots.
- The following are the most common registry functions:
  - *RegOpenKeyEx:* Opens a registry for editing and querying.
  - *RegSetValueEx:* Adds a new value to the registry and sets its data.
  - *RegGetValue:* Returns the data for a value entry in the registry.
- When you see these functions in malware, you should identify the registry key they are accessing.

Analyzing Malicious Windows Programs

- Following figure shows real malware code opening the **Run key** from the registry and **adding** a value so that the program runs each time Windows starts.
- The RegSetValueEx function, which takes six parameters, edits a registry value entry or creates a new one if it does not exist.
- Notes:
  - LEA: Load Effective Address
  - SamDesired: indicates the type of security access requested

Analyzing Malicious Windows Programs



```
0040286F   push     2                 ; samDesired
00402871   push     eax               ; ulOptions
00402872   push     offset SubKey     ; "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402877   push     HKEY_LOCAL_MACHINE ; hKey
0040287C  1call     esi ; RegOpenKeyExW
0040287E   test     eax, eax
00402880   jnz      short loc_4028C5
00402882
00402882 loc_402882:
00402882   lea      ecx, [esp+424h+Data]
00402886   push     ecx               ; lpString
00402887   mov      bl, 1
00402889  2call      ds:lstrlenW
0040288F   lea      edx, [eax+eax+2]
00402893  3push     edx               ; cbData
00402894   mov      edx, [esp+428h+hKey]
00402898  4lea      eax, [esp+428h+Data]
0040289C   push     eax               ; lpData
0040289D   push     1                 ; dwType
0040289F   push     0                 ; Reserved
004028A1  5lea      ecx, [esp+434h+ValueName]
004028A8   push     ecx               ; lpValueName
004028A9   push     edx               ; hKey
004028AA   call     ds:RegSetValueExW
```

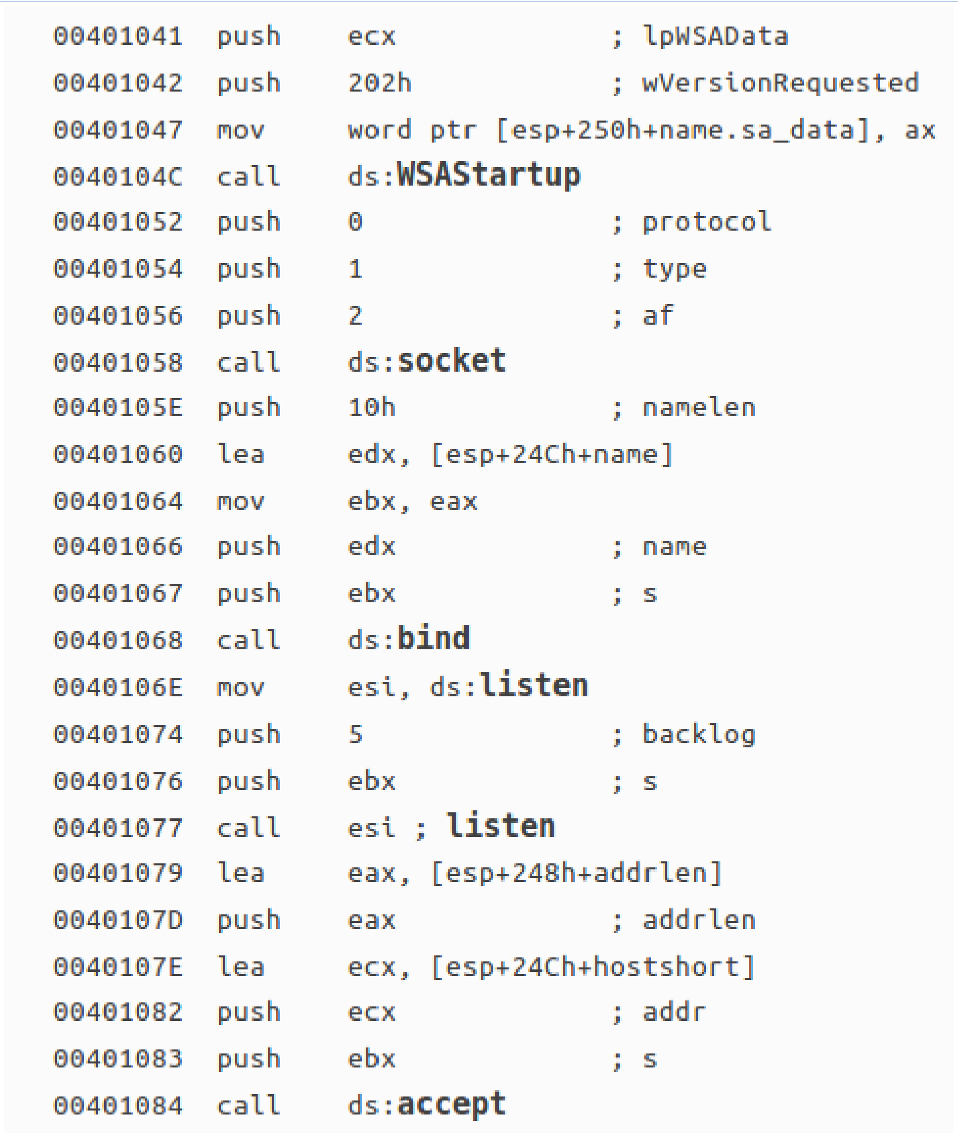
Analyzing Malicious Windows Programs

- Malware commonly relies on network functions to do its dirty work, and there are many Windows API functions for network communication.
- We will understand how to recognize and understand common network functions, so you can identify what a malicious program is doing when these functions are used.

Analyzing Malicious Windows Programs

- Of the Windows network options, malware most commonly uses Berkeley compatible sockets, functionality that is almost identical on Windows and UNIX systems.
- Berkeley compatible sockets' network functionality in Windows is implemented in the Winsock libraries, primarily in ws2_32.dll.
- Of these, the socket, connect, bind, listen, accept, send, and recv functions are the most common.
- **Following figure** shows an example of a server socket program

Analyzing Malicious Windows Programs



```
00401041  push    ecx                ; lpWSAData
00401042  push    202h               ; wVersionRequested
00401047  mov     word ptr [esp+250h+name.sa_data], ax
0040104C  call    ds:WSAStartup
00401052  push    0                  ; protocol
00401054  push    1                  ; type
00401056  push    2                  ; af
00401058  call    ds:socket
0040105E  push    10h                ; namelen
00401060  lea     edx, [esp+24Ch+name]
00401064  mov     ebx, eax
00401066  push    edx                ; name
00401067  push    ebx                ; s
00401068  call    ds:bind
0040106E  mov     esi, ds:listen
00401074  push    5                  ; backlog
00401076  push    ebx                ; s
00401077  call    esi ; listen
00401079  lea     eax, [esp+248h+addrlen]
0040107D  push    eax                ; addrlen
0040107E  lea     ecx, [esp+24Ch+hostshort]
00401082  push    ecx                ; addr
00401083  push    ebx                ; s
00401084  call    ds:accept
```

parag.rughani@gfsu.edu.in

Analyzing Malicious Windows Programs

- Successful reverse engineers **do not evaluate** each instruction individually unless they must.
- The process is just too **tedious**, and the instructions for an entire disassembled program can number in the thousands or even **millions**.
- As a malware analyst, you must be able to obtain a high-level picture of code functionality by analyzing instructions as groups, focusing on individual instructions only as needed.

Analyzing Malicious Windows Programs

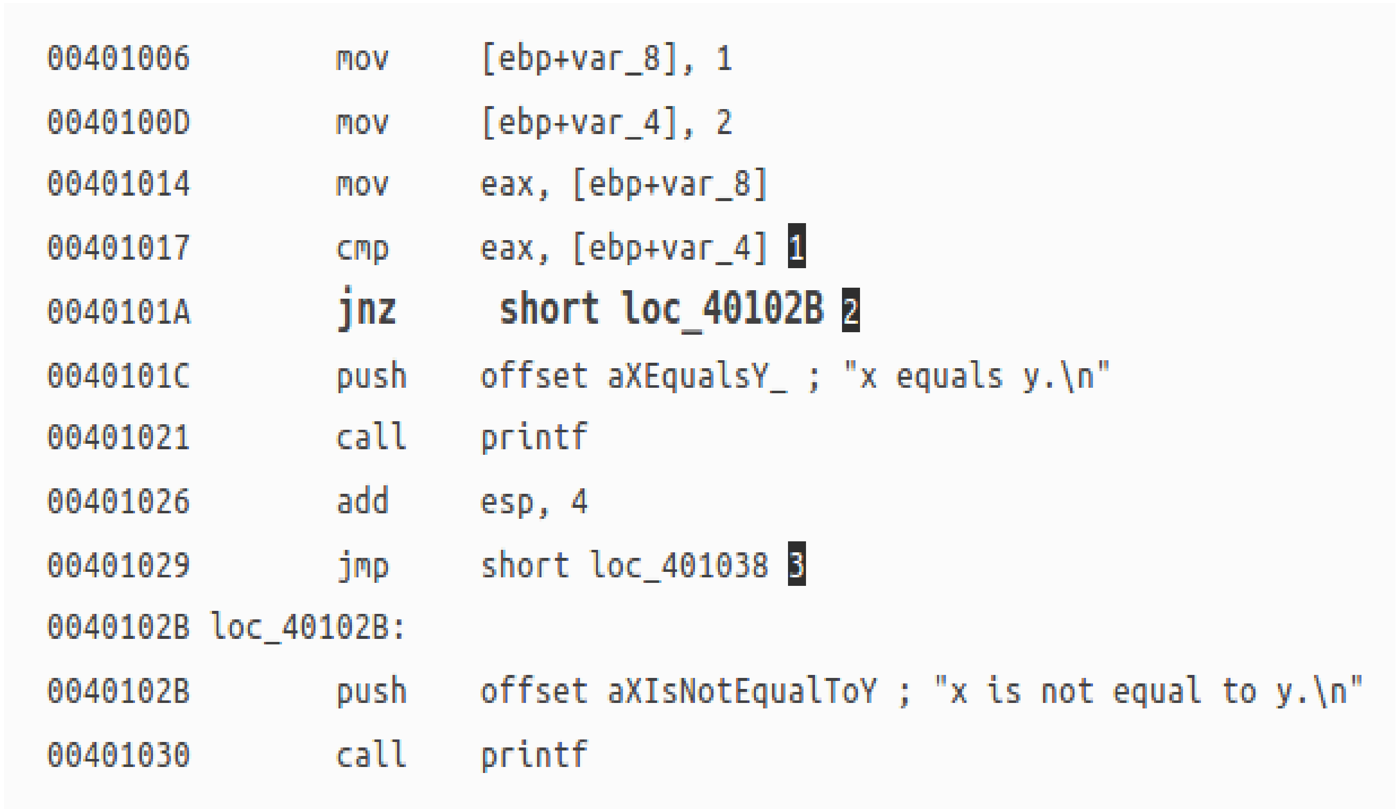
- Malware is typically developed using a **high-level** language.
- A **code construct** is a code abstraction level that defines a functional property but not the details of its implementation.
- Examples of code constructs include loops, if statements, linked lists, switch statements, and so on.
- Programs can be broken down into individual constructs that, when combined, implement the overall functionality of the program.

Analyzing Malicious Windows Programs

- The goal here is to understand the overall functionality of a program, **not** to analyze **every single instruction**.
- Keep this in mind, and don’t get bogged down with the minutiae.
- Focus on the way programs work in general, not on how they do each particular thing.
- Following slides include understanding of various components from a disassembled program.

Analyzing Malicious Windows Programs

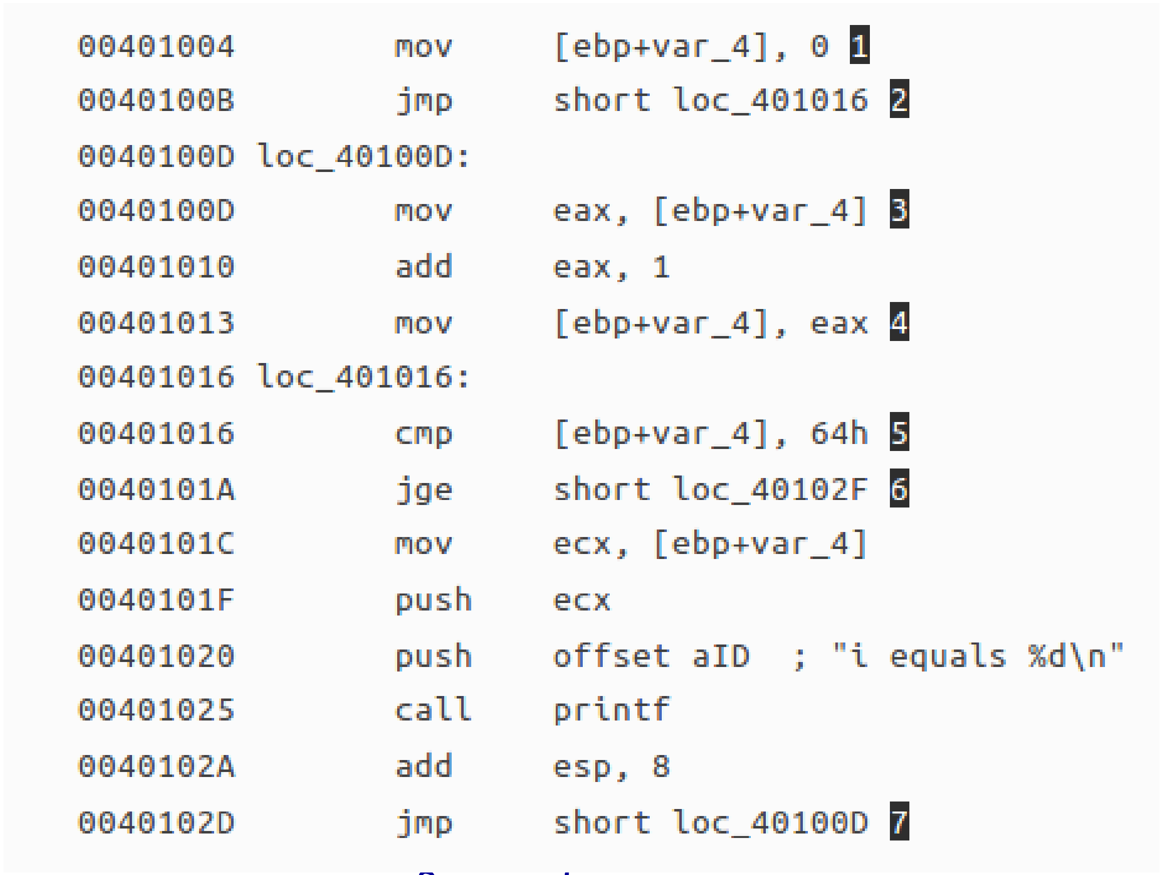
- Conditions: Sample disassembled code looks as follows. (Same can be understood easily in graph mode)



```
00401006        mov     [ebp+var_8], 1
0040100D        mov     [ebp+var_4], 2
00401014        mov     eax, [ebp+var_8]
00401017        cmp     eax, [ebp+var_4] 1
0040101A        jnz      short loc_40102B 2
0040101C        push    offset aXEqualsY_ ; "x equals y.\n"
00401021        call    printf
00401026        add     esp, 4
00401029        jmp     short loc_401038 3
0040102B loc_40102B:
0040102B        push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030        call    printf
```

Analyzing Malicious Windows Programs

- Loops: Sample disassembled code looks as follows. (Same can be understood easily in graph mode)



```
00401004          mov     [ebp+var_4], 0 1
0040100B          jmp     short loc_401016 2
0040100D loc_40100D:
0040100D          mov     eax, [ebp+var_4] 3
00401010          add     eax, 1
00401013          mov     [ebp+var_4], eax 4
00401016 loc_401016:
00401016          cmp     [ebp+var_4], 64h 5
0040101A          jge     short loc_40102F 6
0040101C          mov     ecx, [ebp+var_4]
0040101F          push    ecx
00401020          push    offset aID  ; "i equals %d\n"
00401025          call    printf
0040102A          add     esp, 8
0040102D          jmp     short loc_40100D 7
```

Jge: Jump if Greater or Equal

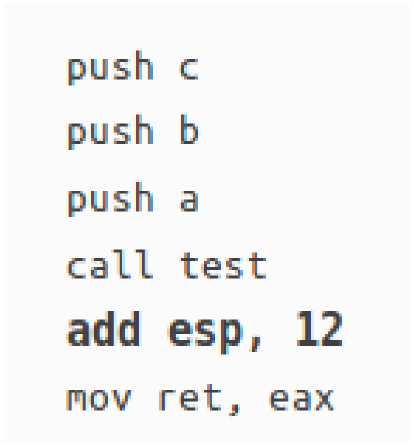
Analyzing Malicious Windows Programs

- Function Calls: The three most common calling conventions you will encounter are cdecl, stdcall, and fastcall.
  - Cdecl: cdecl is one of the most popular conventions. In cdecl, parameters are pushed onto the stack from right to left, the caller cleans up the stack when the function is complete, and the return value is stored in EAX. Next figure will show what the disassembly would look like if following code was compiled to use cdecl.

```
int test(int x, int y, int z);
int a, b, c, ret;
ret = test(a, b, c);
```

Analyzing Malicious Windows Programs

- Function Calls:
  - *Cdecl*: in this segment, **add esp,12** indicates that the **caller** has cleaned the stack.



```
push c
push b
push a
call test
add esp, 12
mov ret, eax
```

Note: The highlighted portion shows that the stack is cleaned up by the ***caller****.*

Analyzing Malicious Windows Programs

- Function Calls:
  - *Stdcall*: The popular stdcall convention is similar to cdecl, except stdcall requires the **callee** to clean up the stack when the function is complete. stdcall is the standard calling convention for the Windows API. Any code calling these API functions will **not need to clean** up the stack, since that's the **responsibility** of the DLLs that implement the code for the API function.
  - *Fastcall:* In fastcall, the first few arguments (typically two) are passed in registers, with the most commonly used registers being **EDX and ECX** (the Microsoft fastcall convention). Additional arguments are loaded from right to left, and the **calling** function is usually responsible for cleaning up the stack, if necessary.

Analyzing Malicious Windows Programs

- Practical:
  - Disassemble any malware and try to get familiarized with it's code. You can refer introduction to x86 assembly instructions slides and IDA Pro slides in combination with slides discussed under this point for better understanding.

Working with OllyDbg

- OllyDbg is an **x86 debugger** and was developed by Oleh Yuschuk.
- OllyDbg provides the ability to analyze **malware** while it is **running**.
- OllyDbg is commonly used by malware analysts and reverse engineers because it's free, it's easy to use, and it has many plug-ins that extend its capabilities.
- It was the primary debugger of choice for malware analysts and exploit developers, until the OllyDbg 1.1 code base was purchased by the Immunity security company and rebranded as **Immunity Debugger** (ImmDbg)

Working with OllyDbg

- If you prefer ImmDbg, don't worry, because it is basically the same as OllyDbg 1.1, and everything discussed in this point applies to both.
- There are several ways to begin debugging malware with OllyDbg.
- You can load executables and even DLLs directly.
- If malware is already running on your system, you can **attach** to the process and debug that way.
- OllyDbg provides a **flexible** system to run malware with command-line options or to execute specific functionality within a DLL.

Working with OllyDbg

- The easiest way to debug malware is to select File ▶ Open, and then browse to the executable you wish to load.
- Once you've opened an executable, OllyDbg will load the binary using **its own loader**.
- This works similarly to the way that the **Windows OS** loads a file.
- By default, OllyDbg will pause at the software developer's entry point, known as **WinMain**, if its location can be determined.
- Otherwise, it will break at the entry point as defined in the **PE header**.

Working with OllyDbg

- In addition to opening an executable directly, you can attach OllyDbg to a running process.
- You’ll find this feature useful when you want to **debug running malware**.
- To attach OllyDbg to a process, select File ▶ Attach.
- Once you are attached with OllyDbg, the current executing thread’s code will be paused and displayed on your screen.

Working with OllyDbg

- As soon as you load a program into OllyDbg, you will see four windows filled with information that you will find useful for malware analysis.
- These windows display information as follows:
  - Disassembler window:
    - This window shows the debugged program's code—the current instruction pointer with several instructions before and after it.
    - Typically, the next instruction to be executed will be highlighted in this window.
    - To modify instructions or data (or add new assembly instructions), press the **spacebar** within this window.

Working with OllyDbg

- :
  - Registers window.
    - This window shows the current state of the registers for the debugged program.
    - As the code is debugged, these registers will change color from black to red once the previously executed instruction has modified the register.
    - As in the disassembler window, you can modify data in the registers window as the program is debugged by right-clicking any register value and selecting Modify.

Working with OllyDbg

- :
  - Stack window:
    - This window shows the current state of the stack in memory for the thread being debugged.
    - This window will always show the top of the stack for the given thread.
    - You can manipulate stacks in this window by right-clicking a stack location and selecting Modify.
    - OllyDbg places useful comments on some stack locations that describe the arguments placed on the stack before an API call.
    - These aid analysis, since you won’t need to figure out the stack order and look up the API argument ordering.

Working with OllyDbg

- :
  - Memory dump window:
    - This window shows a dump of live memory for the debugged process.
    - Press CTRL-G in this window and enter a memory location to dump any memory address. (Or click a memory address and select Follow in Dump to dump that memory address.)
    - To edit memory in this window, right-click it and choose Binary ▶ Edit.
    - This can be used to modify global variables and other data that malware stores in RAM.

Working with OllyDbg

- The **Memory Map** window (View ▶ Memory) displays all memory blocks allocated by the debugged program.
  - The memory map is great way to see how a program is laid out in memory.
- Malware often uses multiple **threads**.
  - You can view the current threads within a program by selecting View ▶ Threads to bring up the Threads window.
  - This window shows the memory locations of the threads and their current status (active, paused, or suspended).

Working with OllyDbg

- You can add or remove a **breakpoint** by selecting the instruction in the disassembler window and pressing F2.
- You can view the **active breakpoints** in a program by selecting View ▶ Breakpoints or clicking the B icon in the toolbar.
- After you close or terminate a debugged program, OllyDbg will typically **save the breakpoint** locations you set, which will enable you to debug the program again with the same breakpoints (so you don't need to set the breakpoints again).

Working with OllyDbg

- In addition to being able to load and attach to executables, OllyDbg can also **debug DLLs**.
- However, since DLLs cannot be executed directly, OllyDbg uses a dummy program called **loaddll.exe** to load them.
- This technique is extremely useful, because malware often comes packaged as a DLL, with most of its code contained inside its **DllMain** function (the initialization function called when a DLL is loaded into a process).
- By default, OllyDbg breaks at the DLL entry point (DllMain) once the DLL is loaded.

Working with OllyDbg

- Tracing is a powerful debugging technique that records detailed execution information.
- OllyDbg supports a variety of tracing features, including:
  - the standard back trace: Any time you are moving through the disassembler window with the Step Into and Step Over options, OllyDbg is recording that movement.
  - call stack trace: You can use OllyDbg to view the execution path to a given function via a call stack trace.
  - and run trace: A run trace allows you to execute code and have OllyDbg save every executed instruction and all changes made to the registers and flags.

Working with OllyDbg

- Practical:
  - Explore various features of OllyDbg.

Kernel Debugging using WinDBG

- WinDbg is a **free debugger** from Microsoft. While not as popular as OllyDbg for malware analysis, WinDbg has many advantages, the most significant of which is kernel debugging.
- Before we begin debugging malicious kernel code, you need to understand **how kernel code works**, why malware writers use it, and some of the unique challenges it presents.
- Windows device drivers, more commonly referred to simply as drivers, allow **third-party** developers to run code in the Windows kernel.

Kernel Debugging using WinDBG

- Drivers are difficult to analyze because they load into memory, stay resident, and respond to requests from applications.
- This is further complicated because applications do **not directly** interact with kernel drivers.
- Instead, they access device objects, which send requests to particular devices.
- Devices are **not necessarily** physical hardware components; the driver creates and destroys devices, which can be accessed from user space.

Kernel Debugging using WinDBG

- In order to system to work properly, drivers must be loaded into the kernel, just as DLLs are loaded into processes.
- When a driver is first loaded, its ***DriverEntry*** procedure is called, similar to DLLMain for DLLs.
- Unlike DLLs, which expose functionality through the **export table**, drivers must register the address for **callback functions**, which will be called when a user-space software component requests a service.
- The DriverEntry routine is responsible for filling this structure in with its **callback functions**.

Kernel Debugging using WinDBG

- Malicious drivers generally do not usually control hardware; instead, they interact with the main Windows kernel components, **ntoskrnl.exe** and **hal.dll**.
- The ntoskrnl.exe component has the code for the core **OS** functions, and hal.dll has the code for interacting with the main **hardware** components.
- Malware will often import functions from one or both of these files in order to **manipulate** the **kernel**.

Kernel Debugging using WinDBG

- Debugging in the kernel is more complicated than debugging a user-space program because when the kernel is being debugged, the OS is **frozen**, and it's impossible to run a debugger.
- Therefore, the most common way to debug the kernel is with Virtual Environment like **VirtualBox**.
- Unlike user-mode debugging, kernel debugging requires a certain amount of initial **setup**.

Kernel Debugging using WinDBG

- You will need to set up the virtual machine to enable kernel debugging, to enable a **virtual serial port** between the virtual machine and the host, and configure WinDbg on the **host** machine.
- Considering that you are running Win7 in Guest machine, you will need to set guest as follows:
  - Open an elevated command prompt
  - bcdedit /debug ON
  - bcdedit /dbgsettings SERIAL DEBUGPORT:1 BAUDRATE:115200

Kernel Debugging using WinDBG

- Create a serial port from Virtual Machine settings with following options:
  - Set named pipe to \\.\pipe\com_1
- Download and install WDK or SDK to use WinDbg (refer: https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit)
- Launch WinDbg
- Select File ▶ Kernel Debug, click the COM tab, and enter the filename and baud rate that you set before (115200 in our case).
- Make sure the Pipe checkbox is checked before selecting OK.

Kernel Debugging using WinDBG

- If the virtual machine is running, the debugger should connect within a few seconds.
- If it is not running, the debugger will wait until the OS boots, and then connect during the boot process.
- Once the debugger connects, consider enabling **verbose** output while kernel debugging, so that you'll get a more complete picture of what is happening.
- With verbose output, you will be notified each time a driver is loaded or unloaded.
- This can help you identify a malicious driver in some cases.

Kernel Debugging using WinDBG

- WinDbg uses a command-line interface for most of its functionality.
- WinDbg's memory window supports memory browsing directly from the command line.
- The ***d*** command is used to read locations in memory such as program data or the stack, with the following basic syntax: ***dx addressToRead***
  - Where x can take any value from: a (ASCII), u (Unicode) or d (32-bit double word)
- The ***e*** command is used in the same way to change memory values. It uses the following syntax: ***ex addressToWrite dataToWrite*** (x takes same values as above)

Kernel Debugging using WinDBG

- Practical:
  - Explore WinDBG

Live Memory Analysis using Volatility

- Refer Unit 1 of Digital Forensics Part – 2 subject.

Anti-Reverse Engineering

- Anti-Reverse Engineering:
  - Anti-Disassembly
  - Anti-Debugging
  - Anti-Virtual Machine Techniques

Anti-Reverse Engineering

- Anti-Disassembly:
  - Anti-disassembly uses specially crafted code or data in a program to cause disassembly analysis tools to produce an incorrect program listing.
  - This technique is crafted by malware authors manually, with a separate tool in the build and deployment process or interwoven into their malware's source code.
  - Malware authors use anti-disassembly techniques to **delay or prevent** analysis of malicious code.
  - Any code that executes successfully can be **reverse-engineered**, but by armoring their code with **anti-disassembly** and **anti-debugging** techniques, malware authors increase the level of skill required of the malware analyst.

Anti-Reverse Engineering

- Anti-Disassembly:
- Jump Instructions with the Same Target
  - The most common anti-disassembly technique seen in the wild is two **back-to-back conditional jump** instructions that both point to the same target.
  - For example, if a jz loc_512 is followed by jnz loc_512, the location loc_512 will **always be jumped to**.
  - The combination of jz with jnz is, in effect, an unconditional jmp, but the disassembler doesn’t recognize it as such because it only disassembles **one instruction at a time**.
  - When the disassembler encounters the jnz, it continues disassembling the false branch of this instruction, despite the fact that it will **never be executed in practice**.

Anti-Reverse Engineering

- Anti-Disassembly:
- A Jump Instruction with a Constant Condition
  - Another anti-disassembly technique commonly found in the wild is composed of a single conditional jump instruction placed where the condition will always be the same.
  - For example, if you put a condition jz after a statement x-x then the statement will always return zero and the condition will be satisfied.

Anti-Reverse Engineering

- Anti-Disassembly:
  - Previous two scenarios represent only when they are improperly disassembled by the first attempt made by the disassembler, but with an interactive disassembler like IDA Pro can produce accurate results.
  - This technique places a data byte strategically after a conditional jump instruction, with the idea that disassembly starting at this byte will **prevent the real instruction that follows from being disassembled** because the byte that is inserted is the opcode for a **multibyte instruction**.
  - This can be called a **rogue (fraudulent)** byte because it is not part of the program and is only in the code to throw off the disassembler.
  - In all of these examples, the rogue byte can be ignored.

Anti-Reverse Engineering

- Anti-Debugging:
  - Anti-debugging is a popular anti-analysis technique used by malware to recognize when it is under the control of a debugger or to thwart debuggers.
  - Malware authors know that malware analysts use debuggers to figure out how malware operates, and the authors use anti-debugging techniques in an attempt to slow down the analyst as much as possible.
  - Once malware **realizes** that it is **running** in a **debugger**, it may **alter** its normal code execution path or modify the code to cause a crash, thus interfering with the analysts' attempts to understand it, and adding time and additional overhead to their efforts.

Anti-Reverse Engineering

- Anti-Debugging:
  - Malware uses a variety of techniques to scan for indications that a **debugger is attached**, including using the Windows API, manually checking memory structure for debugging artifacts, and searching the system for residue left by a debugger. Debugger detection is the most common way that malware performs anti-debugging.
  - The following Windows API functions can be used for anti-debugging:
    - IsDebuggerPresent
    - CheckRemoteDebuggerPresent
    - NtQueryInformationProcess
    - OutputDebugString

Anti-Reverse Engineering

- Anti-Debugging:
  - Recall that debuggers can be used to set breakpoints or to single-step through a process in order to aid the malware analyst in reverse-engineering.
  - However, when these operations are performed in a debugger, they modify the code in the process.
  - Several anti-debugging techniques are used by malware to detect this sort of debugger behavior: **INT scanning, checksum checks, and timing checks**.

Anti-Reverse Engineering

- Anti-Debugging:
  - **INT Scanning:** INT 3 is the software interrupt used by debuggers to temporarily replace an instruction in a running program and to call the debug exception handler— a basic mechanism to set a breakpoint. The opcode for INT 3 is **0xCC**. Whenever you use a debugger to set a breakpoint, it modifies the code by inserting a 0xCC.
  - **Performing Code Checksums:** Malware can calculate a checksum on a section of its code to accomplish the same goal as scanning for interrupts. Instead of scanning for 0xCC, this check simply performs a cyclic redundancy check (CRC) or a MD5 checksum of the opcodes in the malware.

Anti-Reverse Engineering

- Anti-Debugging:
  - **Timing Checks:** Timing checks are one of the most popular ways for malware to detect debuggers because processes run more slowly when being debugged. For example, single-stepping through a program substantially slows execution speed.

Anti-Reverse Engineering

- Anti-Virtual Machine Techniques:
  - Malware authors sometimes use anti-virtual machine (**anti-VM**) techniques to thwart attempts at analysis.
  - With these techniques, the malware attempts to detect whether it is being run inside a virtual machine.
  - If a virtual machine is detected, it can act differently or simply not run.
  - Anti-VM techniques are most commonly found in malware that is widely deployed, such as bots, scareware, and spyware (mostly because honeypots often use virtual machines and because this malware typically targets the average user's machine, which is unlikely to be running a virtual machine).

Anti-Reverse Engineering

- Anti-Virtual Machine Techniques:
  - The popularity of anti-VM malware has been going down recently, and this can be attributed to the great increase in the usage of virtualization.
  - Traditionally, malware authors have used anti-VM techniques because they thought only analysts would be running the malware in a virtual machine.
  - However, today both administrators and users use virtual machines in order to make it easy to rebuild a machine.
  - Malware authors are starting to **realize** that just because a machine is a virtual machine does not necessarily mean that **it isn't a valuable victim**.
  - As virtualization **continues to grow**, anti-VM techniques will probably become even **less common**.

References

1. Practical Malware Analysis by Andrew Honig; Michael Sikorski
2. https://www.virtualbox.org/wiki/Windows_Kernel_Debugging
3. https://msdn.microsoft.com/en-in/library/windows/hardware/dn745912(v=vs.85).aspx