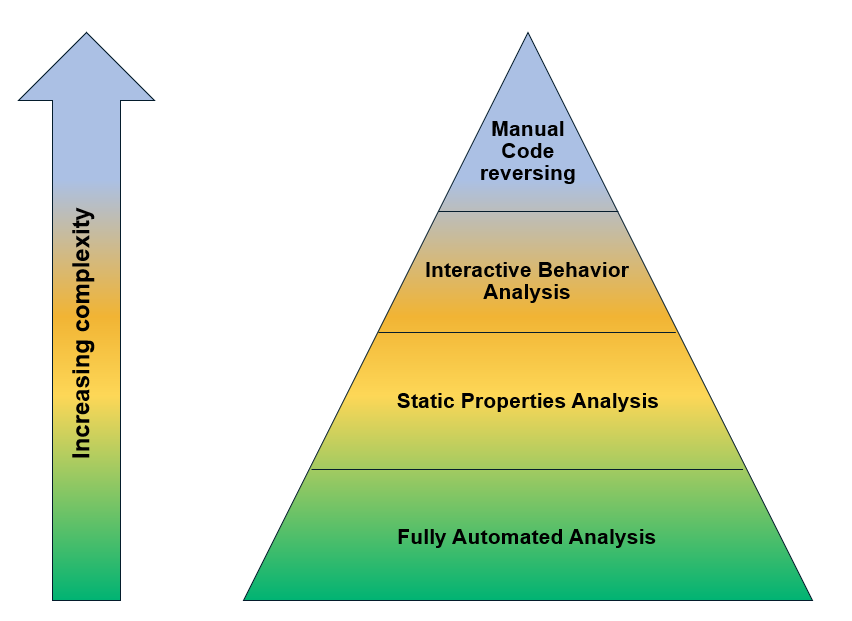
1. **Describe Stages of Malware Analysis.**



* **Fully Automated Analysis :**Automated malware analysis refers to relying on detection models formed by analyzing previously discovered malware samples in the wild. This is the most suited method to process malware at scale and quickly assess the repercussions of a sample on the network infrastructure.
* **Static Properties Analysis :** Static properties analysis involves looking at a file’s metadata without executing the malware. This process is typically something you do within an isolated environment — such as a virtual machine — that’s disconnected from the internet. One of the free tools that you may find useful for this purpose is [PeStudio](https://www.winitor.com/).
* **Interactive Behavior Analysis**: In the next phase, behavior analysis, the malware sample is executed in isolation as the analyst observes how it interacts with the system and the changes it makes. Often, a piece of malware might refuse to execute if it detects a virtual environment or might be designed to avoid execution without manual interaction
* **Manual Code Reversing:** Reverse engineering the code of a sample malware can provide valuable insights. This process can:
* Shed some light on the logic and algorithms the malware uses,
* Expose hidden capabilities and exploitation techniques the malware uses, and
* Provide insights about the communication protocol between the client and the server on the command and control side.

1. **What is a rootkit?**

### A rootkit is a clandestine computer program designed to provide continued privileged access to a computer while actively hiding its presence. The term rootkit is a connection of the two words "root" and "kit."

### Originally, a rootkit was a collection of tools that enabled administrator-level access to a computer or network.

### Root refers to the Admin account on UNIX and Linux systems, and kit refers to the software components that implement the tool. Rootkits are generally associated with malware – such as Trojans, [worms](https://www.veracode.com/security/computer-worm), viruses – that conceal their existence and actions from users and other system processes.

1. **What are CPU registers?**

* Processor registers are normally at the top of the memory hierarchy, and provide the fastest way to access data. The term normally refers only to the group of registers that are directly encoded as part of an instruction, as defined by the instruction set.
* So that it can store information (under different values and different sizes), each processor is composed of different parts, kind of “boxes”, called *registers.* They constitute one of the most important parts of the CPU, and according to the characteristics of the information to store (value, size, etc.) , using registers instead of memory makes the processor faster. We can consider three kinds of registers:

1. *General Registers:* Used to manipulate data, to pass parameters when calling a DOS function, and to store intermediate results
2. *Status Registers.*
3. *Segment Register:* Used to store the starting address of a segment. It may be the address of the beginning of a program’s instructions, the beginning of data, or the beginning of the stack.

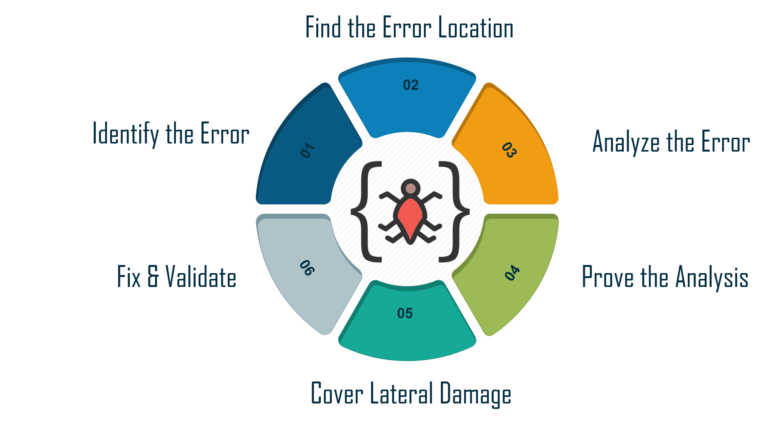
* Almost all registers can be divided into 16 and 8 bits. General registers begin with the letters A, B, C and D, and are the most used registers.
* *AX – Accumulator Register*: used to perform arithmetic operations or send a parameter to an interruption.
* *BX – Base Register:* used to perform arithmetic operations or as the base address of an array.
* *CX – Counter Register*: used generally as a counter on loops.
* *DX – Data Register*: used to store data for functions, and as a port number in input / output operations.
* AX, BX, CX and DX are 16-bit-registers

1. **Explain C code construct in Assembly.**

* Understanding of C code construct in Assembly The basic aspects of a programming language (in this case C) and shows how they are implemented in Assembly to help an analyst pick out the patterns more easily.
* It covers some of the concepts I’m already familiar with, at least at a basic level, such as if statements, loops, and arrays, but also adds a little more complexity with structs and linked lists.
* look up some of the C syntax for structs and linked lists to get a better idea of why they’re formatted

1. **Describe Encode, Encryption and Hash.**

* Encoding: Is the process of transforming data( in our case plaintext) to another format that can be easily and safely used by a different system
* *Encryption* is a process to convert the information into a cipher using keys, to maintain the confidentiality.
* Hashing: A Hash is basically a string that is generated from the input string by passing it through a *Hash Algorithm*. This hashed string is always of a fixed length no matter what is the size of the input string. Hashing can also be considered as *“One Way Encryption”.*
  + 1. Hashing Algorithms are:  
       - Message Digest (MD) Algorithm  
       - Secure Hash Algorithm (SHA)
    2. - RACE Integrity Primitives Evaluation Message Digest (RIPEMD)  
       - Whirlpool  
       - Cyclic Redundancy Check (CRC)

1. **What is debugging with stages?** 
   * **Identify the Error:**A bad identification of an error can lead to wasted developing time. It is usual that production errors reported by users are hard to interpret and sometimes the information we receive is misleading. It is import to identify the actual error.
   * **Find the Error Location:**After identifying the error correctly, you need to go through the code to find the exact spot where the error is located. In this stage, you need to focus on finding the error instead of understanding it.
   * **Analyze the Error:**In the third step, you need to use a bottom-up approach from the error location and analyze the code. This helps you in understanding the error. Analyzing a bug has two main goals, such as checking around the error for other errors to be found, and to make sure about the risks of entering any collateral damage in the fix.
   * **Prove the Analysis:**Once you are done analyzing the original bug, you need to find a few more errors that may appear on the application. This step is about writing automated tests for these areas with the help of a test framework.
   * **Cover Lateral Damage:**In this stage, you need to create or gather all the unit tests for the code where you are going to make changes. Now, if you run these unit tests, they all should pass.
   * **Fix & Validate:** The final stage is the fix all the errors and run all the test scripts to check if they all pass.

Here is a list of some of the widely used debuggers:

* Radare2, WinDbg, Valgrind

1. **What is dynamic analysis?**

* Executing a given malware sample within a controlled environment and monitoring its actions in order to analyze the malicious behaviour is called dynamic malware analysis.
* Since Dynamic Malware Analysis is performed during runtime and malware unpacks itself, dynamic malware analysis evades the restrictions of static analysis (i.e., unpacking and obfuscation issues).
* Thereby it is easy to see the actual behaviour of a program.
* Another major advantage is that it can be automated thus enabling analysis at a large scale basis. However, the main drawback is so-called dormant code: That is, unlike static analysis, dynamic analysis usually monitors only one execution path and thus suffers from incomplete code coverage.
* In addition there is the danger of harming third party systems, if the analysis environment is not properly isolated or restricted respectively. Furthermore, malware samples may alter their behaviour or stop executing at all once they detect to be executed within a controlled analysis environment.

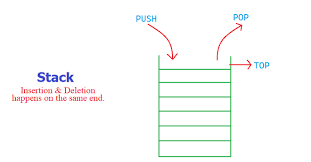
Mainly two basic approaches for dynamic malware analysis can be distinguished:

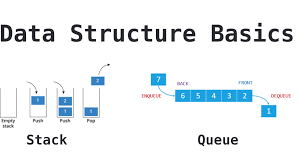
• Analyzing the difference between defined points: A given malware sample is executed for a certain period of time and afterwards the modifications made to the system are analyzed by comparison to the initial system state. In this approach, Comparison report states behaviour of malware.[8]

• Observing runtime-behavior: In this approach, malicious activities launched by the malicious application are monitored during runtime using a specialized tool.

1. **Explain Stack memory.**

The stack is a memory area that can hold temporary data (functions parameters, variables, etc.) and is designed to behave in a “Last In, First Out” context, which means the first value stored in the stack (or pile) will be the last entry out. The sample always given when it comes to explaining how the stack works is “plates stacked up to be washed”; the last to be stacked will be the first to be washed.





1. **What is threat intelligence?**

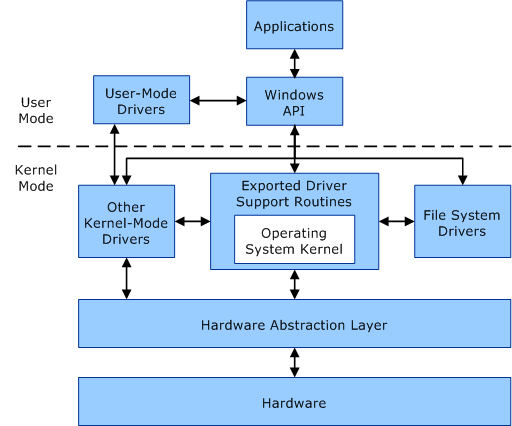
Threat intelligence, or cyber threat intelligence, is information an organization uses to understand the threats that have, will, or are currently targeting the organization. This info is used to prepare, prevent, and identify cyber threats looking to take advantage of valuable resources.

## Indicators of Compromise

* IP addresses, URLs and Domain names: An example would be malware targeting an internal host that is communicating with a known threat actor.
* Email addresses, email subject, links and attachments: An example would be a phishing attempt that relies on an unsuspecting user clicking on a link or attachment and initiating a malicious command.
* Registry keys, filenames and file hashes and DLLs: An example would be an attack from an external host that has already been flagged for nefarious behaviour or that is already infected.

1. **What is assembly debugger?**

* The debugger automatically displays the contents of memory locations and registers as they are accessed and displays the address of the program counter.
* This display makes assembly debugging a valuable tool that you can use together with source debugging



## **User Mode**

The system is in user mode when the operating system is running a user application such as handling a text editor. The transition from user mode to kernel mode occurs when the application requests the help of operating system or an interrupt or a system call occurs.

**Debugger Features**

* Examine Variables.
* Thread Window.
* Stack Window.
* Watch Window.
* Log Window.
* Profiler.
* Real time comment-out.
* Edit Breakpoint Window.

1. **Describe Types of Breakpoints.**

* Breakpoints are temporary markers that you place in your executable program to tell the debugger to stop your program at a given point
* Breakpoints don't necessarily need special hardware features. Debugger here relies on modifying original binary (it's copy that is loaded to memory). When you set a breakpoint, debugger will place special instruction at the location of breakpoint.
* This special instruction needs to somehow let debugger detect when it (this special instruction) is executing.
* This can be some instruction that causes some kind of interrupt/exception, that debugger can hook onto, or some instruction that handles the control to debug unit.
* If this runs under some OS, that OS needs to support modifying running program (with something like trace poke/peek).

**Types of Breakpoints**

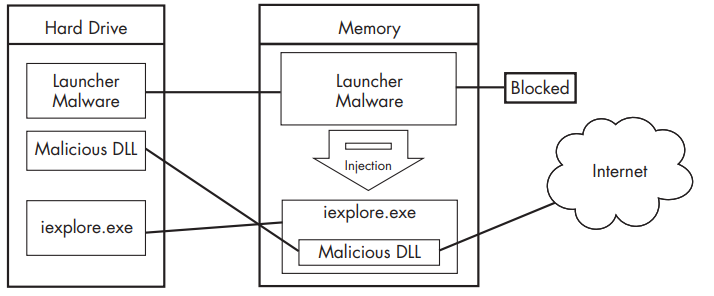
* Software breakpoints:- software execution breakpoints, which cause a program to stop when a particular instruction is executed. When you set a breakpoint without any options, most popular debuggers set a software execution breakpoint by default. The debugger implements a software breakpoint by overwriting the first byte of an instruction with 0xCC, the instruction for INT 3, the breakpoint interrupt designed for use with debuggers. When the 0xCC instruction is executed, the OS generates an exception and transfers control to the debugger.
* Hardware breakpoints:- the x86 architecture supports hardware execution breakpoints through dedicated hardware registers. Every time the processor executes an instruction, there is hardware to detect if the instruction pointer is equal to the breakpoint address. Unlike software breakpoints, with hardware breakpoints, it doesn’t matter which bytes are stored at that location. Hardware breakpoints have another advantage over software breakpoints in that they can be set to break on access rather than on execution.
* Conditional breakpoints:- Conditional breakpoints are software breakpoints that will break only if a certain condition is true. Conditional breakpoints are implemented as software breakpoints that the debugger always receives. The debugger evaluates the condition, and if the condition is not met, it automatically continues execution without alerting the user
* Breakpoints on memory

1. **Explain Malware Process Injection.**

* The most popular covert launching technique is process injection.
* As the name implies, this technique injects code into another running process, and that process unwittingly executes the malicious code.
* Malware authors use process injection in an attempt to conceal the malicious behavior of their code, and sometimes they use this to try to bypass host-based firewalls and other process-specific security mechanisms.
* Certain Windows API calls are commonly used for process injection.

**DLL Injection**

* DLL injection—a form of process injection where a remote process is forced to load a malicious DLL—is the most commonly used covert loading technique.
* DLL injection works by injecting code into a remote process that calls LoadLibrary, thereby forcing a DLL to be loaded in the context of that process.
* Once the compromised process loads the malicious DLL, the OS automatically calls the DLL’s DllMain function, which is defined by the author of the DLL.
* This function contains the malicious code and has as much access to the system as the process in which it is running.
* Malicious DLLs often have little content other than the Dllmain function, and everything they do will appear to originate from the compromised process.

****

**Direct Injection**

* Like DLL injection, direct injection involves allocating and inserting code into the memory space of a remote process. Direct injection uses many of the same Windows API calls as DLL injection.
* The difference is that instead of writing a separate DLL and forcing the remote process to load it, direct injection malware injects the malicious code directly into the remote process.
* Direct injection is more flexible than DLL injection, but it requires a lot of customized code in order to run successfully without negatively impacting the host process.
* This technique can be used to inject compiled code, but more often, it’s used to inject shellcode.
* Three functions are commonly found in cases of direct injection: VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread. There will typically be two calls to VirtualAllocEx and WriteProcessMemory. The first will allocate and write the data used by the remote thread, and the second will allocate and write the remote thread code. The call to CreateRemoteThread will contain the location of the remote thread code (lpStartAddress) and the data (lpParameter)

1. **Explain YARA Signature.**

YARA is a tool designed to help malware researchers identify and classify malware samples. It’s been called the [pattern-matching Swiss Army knife](https://plusvic.github.io/yara/) for security researchers (and everyone else). It is multiplatform and can be used from both its command-line interface or through your own Python scripts.

The tool allows you to conduct signature-based detection of malware, something similar to what antivirus solutions can do for you.

There is support for three different types of strings:

* Hexadecimal strings, which are useful for defining raw bytes;
* Text strings;
* Regular expressions.

1. **What is an obfuscation in malware attack?**

* Malware authors often use packing or obfuscation technique to make their files more difficult to detect or analyze.  Malware obfuscation is a technique used to create textual and binary data difficult to interpret. It helps adversaries to hide critical strings in a program, because they reveal patterns of the malware’s behavior. The strings would be registry keys and infected URLs.
* Packed programs are a subset of obfuscated programs in which the malicious program is compressed and cannot be analyzed. Packer and obfuscation techniques will limit the attempts to statically analyze the malware.
* Non-Malicious programs always include many strings. Malware that is packed or obfuscated contains very few strings. If the program has only few strings, it is probably either obfuscated or packed, which gives a clue that it may be malicious.

Some common obfuscation techniques include the following:

* **Renaming.** The obfuscator alters the methods and names of variables. The new names may include unprintable or invisible characters.
* **Packing.** This [compresses](https://www.techtarget.com/searchstorage/definition/compression) the entire program to make the code unreadable.
* **Control flow.** The decompiled code is made to look like [spaghetti logic](https://www.techtarget.com/searchsoftwarequality/tip/Fix-spaghetti-code-and-other-pasta-theory-antipatterns), which is unstructured and hard to maintain code where the line of thought is obscured. Results from this code are not clear, and it's hard to tell what the point of the code is by looking at it.
* **Instruction pattern transformation.** This approach takes common instructions created by the compiler and swaps them for more complex, less common instructions that effectively do the same thing.
* **Dummy code insertion.** Dummy code can be added to a program to make it harder to read and reverse engineer, but it does not affect the program's logic or outcome.
* **Metadata or unused code removal.** Unused code and metadata give the reader extra information about the program, much like annotations on a Word document, that can help them read and [debug](https://www.techtarget.com/searchsoftwarequality/definition/debugging) it. Removing metadata and unused code leaves the reader with less information about the program and its code.
* **Opaque predicate insertion.** A predicate in code is a logical expression that is either true or false. Opaque predicates are conditional branches -- or if-then statements -- where the results cannot easily be determined with statistical analysis. Inserting an opaque predicate introduces unnecessary code that is never executed but is puzzling to the reader trying to understand the decompiled output.
* **Anti-debug.** Legitimate software engineers and hackers use debug tools to examine code line by line. With these tools, software engineers can spot problems with the code, and hackers can use them to reverse engineer the code. IT security pros can use anti-debug tools to identify when a hacker is running a debug program as part of an attack. Hackers can run anti-debug tools to identify when a debug tool is being used to identify the changes they are making to the code.
* **Anti-tamper.**These tools detect code that has been tampered with, and if it has been modified, it stops the program.
* **String encryption.** This method uses encryption to hide the strings in the executable and only restores the values when they are needed to run the program. This makes it difficult to go through a program and search for particular strings.
* **Code transposition.** This is the reordering of routines and branches in the code without having a visible effect on its behavior.

**Debugger**: Debugger is a process used to remove bugs from code. It permits testing and debugging other programs. Sometime, it also provides two modes of operations. It is used to prevent incorrect operation of software or system. It also uses instruction-set simulators instead of running a program directly on processor to achieve higher level of control over its execution.

**The basic steps in debugging are:**

* Identify that a bug exists.
* Isolate the source of the bug.
* Identify the cause of the bug.
* Determine a fix for the bug.
* Apply the fix and test it.

**What is Source-level debuggers?**

* Source-level debuggers allow you to set breakpoints, which stop on lines of source code, in order to examine internal variable states and to step through program execution one line at a time.
* The main difference between compiler and debugger is that compiler converts the source code to alike machine code to execute the tasks defined in the program while debugger helps to recognize the errors of a program and to fix them

**Key Differences Between Stack and Heap Allocations** 

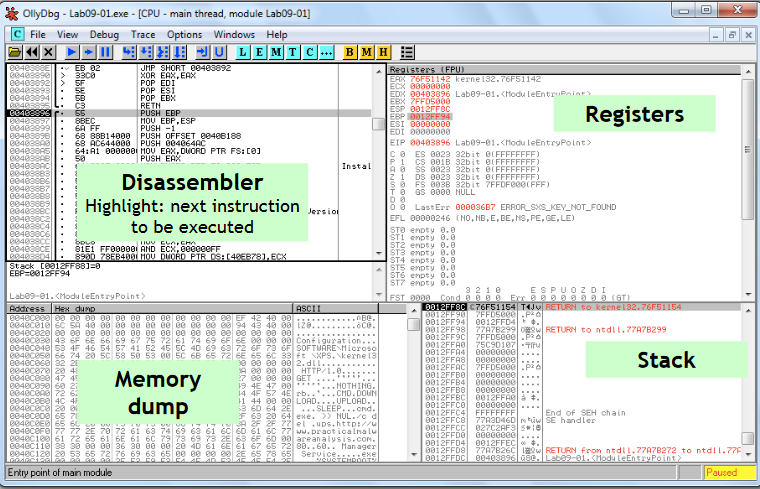
1. In a stack, the allocation and de-allocation are automatically done by the compiler whereas in heap, it needs to be done by the programmer manually.
2. Handling of Heap frame is costlier than the handling of the stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
4. Stack frame access is easier than the heap frame as the stack has a small region of memory and is cache-friendly, but in case of heap frames which are dispersed throughout the memory so it causes more cache misses.
5. A stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
6. Accessing time of heap takes is more than a stack.

**Working with OllyDbg and Immunity Debugger:**

File, Open  
• Add command-line arguments if needed  
• OllyDbg will stop at the entry point, WinMain, if it can be determined  
• Otherwise it will break at the entry point defined in the PE Header  
– Configurable in Options, Debugging Options

Attaching to a Running Process

* File, Attach
* OllyDbg breaks in and pauses the program and all threads
  + If you catch it in DLL, set a breakpoint on
  + access to the entire code section to get to the interesting code
  + Ctrl+F2 reloads the current executable
  + F2 sets a breakpoint



**Kernel debugging:**

Kernel is performed on two systems because there is only one kernel; if the kernel is at a breakpoint, no applications can be running on the system. One system runs the code that is being debugged, and another runs the debugger.

A kernel-mode debugging environment **typically has two computers: the host computer and the target computer**. The debugger runs on the host computer, and the code being debugged runs on the target computer. The host and target are connected by a debug cable.

**Kernel Mode:**

* Kernel Mode is a privileged mode of operation in which processes can execute within the Microsoft Windows NT and Windows 2000 operating systems. Processes running in kernel mode can access system memory and hardware.
* The system starts in kernel mode when it boots and after the operating system is loaded, it executes applications in user mode. There are some privileged instructions that can only be executed in kernel mode.
* These are interrupt instructions, input output management etc. If the privileged instructions are executed in user mode, it is illegal and a trap is generated.

**Kernel Debugging with WinDBG :**

WinDbg is a kernel-mode and user-mode debugger that is included in Debugging Tools for Windows.

## Set up a kernel-mode debugging

A kernel-mode debugging environment typically has two computers: the host computer and the target computer. The debugger runs on the host computer, and the code being debugged runs on the target computer. The host and target are connected by a debug cable.

The Windows debuggers support these types of cables for debugging:

* USB 2.0 / USB 3.0
* Serial (also called null modem)
* Ethernet

Command Prompt setup

* In a Command Prompt window, you can initiate a kernel-mode debugging session when you launch WinDbg. Enter one of the following commands:
* windbg [-y SymbolPath] -k net: port=PortNumber, key=Key [, target=TargetIPAddress|TargetMachineName]
* windbg [-y SymbolPath] -k usb:targetname=USBString
* windbg [-y SymbolPath] -k com:port=ComPort,baud=BaudRate
* windbg [-y SymbolPath] -k com:ipport=SerialTcpIpPort,port=SerialIPAddress
* windbg [-y SymbolPath] -k com:pipe,port=\\VMHost\pipe\PipeName[,resets=0][,reconnect]
* windbg [-y SymbolPath] -k com:modem
* windbg [-y SymbolPath] -kl
* windbg [-y SymbolPath] -