Atharva Mahesh Gokhale
MSc. Cyber Security
ERN: 0323003000080002027
Malware Analysis TA-2

Q1. How to detect a packer? Explain any 3 methods.

A packer is a program that takes the executable file as an input, and it uses compression to obfuscate the executable's content. The obfuscated content is then stored within the structure of a new executable file; the result is a new executable file (packed program) with obfuscated content on the disk.

Detecting packers, which are tools used to compress, encrypt, or modify executable files, is essential in cybersecurity for identifying potentially malicious software. Out of many, the three prominent methods to detect these packers are:

1. Signature based detection
   - Signatures are unique patterns or sequences of bytes that identify specific packers
   - Antivirus softwares and IDS use signature databases to match known packer signature against files.
   - If a file matches a known packer signature, its flagged as potentially suspicious.
   - However, this method relies on having signatures for all known packers, so it may not detect newer customised packers.
2. Static Analysis:
   - Involves examining the characteristics of the file without executing it.

- Look for irregularities in the file structure or code that indicating packing.
- Common signs include unusually small file size, lack of typical header information and presence of specific sections like a code cave or overlay.
- Tools like PEiD and ExeInfo PE are commonly used for static analysis to identify packers based on file structure and metadata.

3. Dynamic Analysis:
- Involves executing the file in a controlled environment (sandbox) to observe its behaviour
- Monitor system calls, API calls, memory usage during execution.
- Look for patterns indicative of unpacking routines such as dynamically allocating memory, decrypting sections or loading additional modules.
- Behavioural analysis can reveal packer's behaviour and its potential malicious intent.
- Tools like IDA Pro, OllyDbg, and Procmon are commonly used for dynamic analysis to trace execution flow and monitor system activity

Q2. How to analyze DLL (Dynamic Link Libraries) and run-time link libraries?

- Analyzing DLL and RTLL involves examining their structure, functions, dependencies and behaviour.

1. Static Analysis:
- File inspection: Start by examining the DLL file using tools like Dependency Walker, PE Explorer or IDA Pro. Look at the

file headers, section, imports and exports to understand its structure.

- Dependency Analysis : Identify the dependency of DLL, including other DLLs it relies on. Understanding these dependencies can provide insight into the functionality and purpose of the DLL.
- Exported Functions : Analyze the functions exported by the DLL. These are the functions that other programs can call. Understanding the exported functions helps in understanding the capabilities of the DLL.
- Strings Analysis : Extract and analyze strings embedded within the DLL. These strings may contain important information such as function names, error messages or hardcoded URLs.

2. Dynamic Analysis :

- Dynamic Loading : Run the DLL in a controlled env or debugger to observe its behaviour. Monitor API calls file system activity, network communication and memory usage.
- Function Hooking : Use tools like API monitor or Frida to intercept and analyze function calls made by the DLL at the runtime. This can help in understanding the DLLs functionality and interactions with the system.
- Memory Analysis : Analyze the memory allocated by the DLL during runtime. Look for signs of malicious behaviour such as code injection, heap spraying or buffer overflows.

3. Behavioral Analysis:

- Execution Flow: Trace the execution flow of the DLL to understand how it operates. Identify entry points, control flow structures and any obfuscation techniques used
- Error Handling - Analyze how the DLL handles errors and exceptions. Look for error messages, logging or abnormal termination behaviour.
- Resource Usage: Monitor resource utilization such as CPU, memory and network activity. Unusual spikes or patterns may indicate malicious activity.

4. Code Reversing:

- Disassembly: Disassemble the DLL code using disassembler like IDA pro or Ghidra. Analyze the assembly code to understand the logic and algorithms used.
- Function Identification: Identify analyze individual fn within the DLL. Reverse engineer their purpose and functionality.
- Decompilation: Use decompilation tools like Hex-Rays (IDA-Pro plugin) or RetDec to generate higher level code from the assembly. This can make the analysis process easier and more intuitive.

Q3. Explain the unpacker stub and unpacking methods in detail.

An unpacker stub, also known as an unpacker routine is a piece of code embedded within a packed executable that is responsible for dynamically decompressing or decrypting the original code and data. The purpose of

the unpacker stub is to restore the packed executable to its original form, making the unpacked code executable by the OS.

1. Unpacking Stub:
- The unpacker stub is typically a small portion of code embedded within the packed executable.
- Its primary function is to perform the unpacking process, which involves decompressing or decrypting the packed data and code
- Unpacking stubs are often designed to be small and efficient, as they need to fit within the constraints of the packed executable.
- Unpacking stubs can be simple or complex, depending on the sophistication of the packer and the level of obfuscation used.

2. Unpacking Methods:
- Runtime Unpacking: This method involves executing the packed executable in a controlled environment, such as a debugger or sandbox. The unpacker stub detects when the program is running and dynamically unpacks the original code into memory before executing it. Runtime unpacking ~~it~~ makes it difficult for static analysis tools to analyze the packed executable because the original code wasn't present on the disk.
- Emulation based Unpacking: In this method, the packed executable is emulated in a virtual environment, and the behaviour of the unpacker stub is visualized and analyzed without actually executing the code on the host system. Emulation based unpacking allows for a

deeper analysis of the unpacking process without the risk of running potentially malicious code on the host machine.

- Static unpacking - It involves analyzing the packed executable without executing it. This method typically requires reverse engineering techniques to identify the unpacker stub, understand its functionality, and extract the original code and data. Static unpacking can be challenging, especially if the packer employs advanced obfuscation techniques to hide the unpacker stub.
- Manual Unpacking: It is a labour-intensive process that involves manually analyzing the packed executable and extracting the original code and data. This process may require disassembling the packed executable, identifying the unpacker stub, and tracing its execution to understand how it unpacks the data. Manual unpacking can be time consuming and requires a deep understanding of assembly language and reverse engineering techniques.

Q4. Setup an Ideal lab for dynamic malware analysis.

Setting up an ideal lab for dynamic analysis involves creating a controlled environment where we can only safely execute and analyze malicious software

1. Isolation:

- Using a dedicated physical or virtual machine for malware analysis to prevent contamination of your primary system
- Ensuring that the analysis machine is not connected to any production networks to prevent the spread of the

malware.

- Consider using network segmentation or VLAN to isolate the system from other networked systems.

2. Virtualization:

- Use virtualization softwares like VMware, oracle VirtualBox or Hyper-V to create and manage isolated analysis environments.
- Take snapshots of the clean state of the virtual machine (VMs) before executing malware to facilitate easy rollback and recovery.
- Utilize features like n/w segmentation, virtual switches and vm clones to create complex n/w configurations for analyzing malware behaviour in diff n/w env.

3. monitoring and logging:

- Incorporate monitoring and logging tools to capture system activities, n/w traffic, file system changes and registry modifications.
- Use tools like - Sysinternals suite. (Tcpdump, procmon) wireshark, snort can help monitor the system and n/w activities during malware execution
- Configure logging tools to capture detailed information about the malware behaviour, such as API calls, process creation, and file modifications.

4. Sandbox and Emulators:

- Setup sandbox or emulators to execute malware in a controlled environment and observe its behaviour.
- Tools like Cuckoo sandbox, Joe Sandbox, Hybrid Analysis provide automated malware analysis capabilities, including dynamic analysis in sandbox environments.

5. Traffic Analysis:

- Use network analysis tools like wireshark or security onion to capture and analyse n/w traffic generated by the malware.
- Analyse the traffic and n/w packets to identify communication protocols, command and control (C2) traffic, and data exfiltration attempts.

- Other tools related to behavioral analysis, dynamic instrumentation and remidiation - cleanup can be added to make our malware analysis lab more functional and rigorous.

Q5. Illustrate the importance of malware analysis in the industrial aspects.

Malware analysis plays a crucial role in various industrial aspects, primarily in the cybersecurity and threat intelligence. Here are several key points illustrating its importance:

1. Threat detection and prevention:

- Malware Analysis (MA) helps organizations detect and prevent malicious software from compromising their systems and networks.
- By analyzing malware samples, security teams can identify new threats, understand their behaviour and develop countermeasure to protect against them
- Proactively analyzing malware allows organizations to stay ahead of evolving threats and vulnerabilities reducing the risk of cyber attacks.

2. Incident Response and Forensics:

- In the event of a security breach or incident, malware analysis is essential for conducting forensic investigations.
- Security analysts analyze malware samples to determine how attackers gained access, what actions they performed, and the extent of the damage.
- Malware analysis provides valuable insights into the tactics, techniques, and procedures (TTPs) used by the threat actors, aiding in incident response efforts and remediation.

3. Security Product Developement

- Malware analysis serves as a foundation for developing and enhancing security products and solutions.
- Security vendors leverage malware samples to train machine learning models, improve detection algorithms and enhance threat intelligence feeds.
- Analyzing malware helps security product developers understand emerging threats and design effective defenses to mitigate them.

4. Threat Intelligence and Sharing

- Malware analysis generates valuable threat intelligence that can be shared with the broader cybersecurity community.
- Threat intelligence feeds containing information about malware families, IOCs, and attack techniques enable organizations to better defend against threats.
- Sharing malware analysis findings fosters collaborations

among security professionals, enhances situational awareness, and strengthens collective defenses against cyber threats.

5. Compliance and Risk management:

- Malware analysis essential for organizations to meet regulatory compliance requirements and manage cybersec risks effectively.
- Industries such as finance, healthcare, government are subject to strigent regulatory standards that mandate proactive measures to protect against malware and cyber threats.
- Conducting malware analysis helps organizations demonstrate due diligence in safeguarding sensitive data and systems, thereby reducing legal and financial risks associated with non-compliance and security breaches.

Q6. Explain PE Header and its sections.

The portable Executable (PE) file format is the standard file format for executables, object code, DLLs in 32-bit and 64-bit versions of the windows OS. The PE header and its sections play a vital role in organizing and describing the contents of a PE file.

1. PE Header:

- The PE Header is a data structure located at the beginning of a PE file that contains essential information about the file format and its components
- It typically starts with a signature, which is the

ASCII string "PE\0\0"

- The PE header includes various fields and structures that provide the metadata about the PE file, such as the machine architecture, the number of sections, the entry point address, and more
- Key components of the PE Header include;

a. DOS Header: The DOS header proceeds the PE header and contains information for MS-DOS compatibility including the DOS stub program

b. File Header: This section includes information about the overall organization of the PE file, such as the target machine architecture, the number of sections and the timestamps of compilation.

c. Optional Header: The optional header contains additional information about the PE file such as the image base address, entry point address, section alignment, and subsystem type.

d. Data Directories: These directories contain pointers to various data structures such as the import table export table, resource table, and the debug info.

2. PE Header Sections:

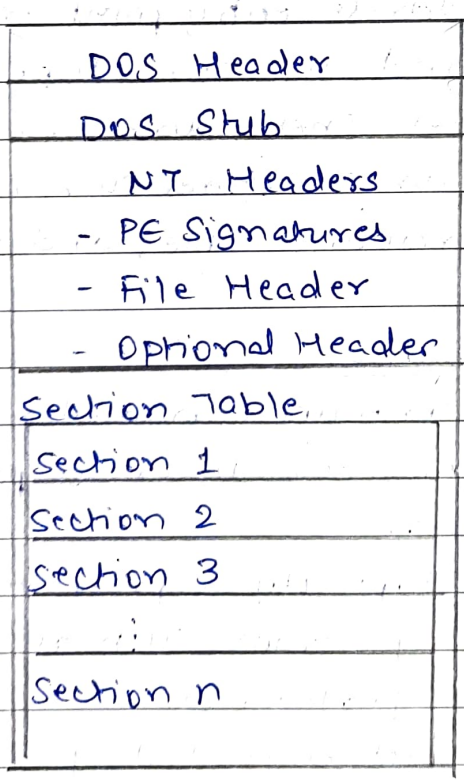
- Sections are contiguous, named regions within a PE file that organize different parts of the executable code and data.
- Each section typically represents a different aspect of the program, such as code, data resources and import/export tables.
- Sections have attributes that specify their characteristics, such as whether they contain executable code,

initialized data, uninitialized data or read only data.

- common section names include

a. .text - contains executable code.
b. .data - contains initialized data.
c. .rsrc - contains resources such as icons, dialogs, and version information.
d. .idata - contains the import table, which lists external functions imported by the executable.

- Sections are aligned based on the specified alignment value in the optional header to optimize loading and execution performance.
- The layout and characteristics of the sections can vary depending on the compiler settings, linker options, and purpose of the executable.



Structure of a PE Header.

Q7. Discuss Pros and cons of Sandbox

Sandboxes are controlled environment used to execute and analyze potentially malicious software safely

1. Pros of Sandboxes.

a. Security Isolation:

- Sandboxes provide controlled environment that isolates potentially harmful code from the rest of the system. This isolation prevents the malware from spreading to other parts of the system or accessing sensitive data.

b. Safe execution:

- Sandboxes allow security researchers to execute suspicious files or programs safely. By running the malware in a sandbox, researchers can observe its behavior without risking damage to the host system.

c. Behavioral Analysis:

- Sandboxes capture and analyze the behavior of malware during execution. This analysis helps in understanding the capabilities, intentions, and potential impact of the malware.

d. Automated Analysis:

- Many sandboxes offer automated analysis capabilities allowing for the quick and efficient evaluation of multiple malware samples. Automated analysis tools can generate reports, extract indicators of compromise (IOC), and provide insights into emerging threats.

e. Network monitoring:

- Sandboxes include network monitoring capabilities

to capture and analyze network traffic generated by the malware. This helps in identifying the command and control center (C2), data exfiltration, and other malicious activities.

f. Threat Intelligence.

- Sandboxes generate valuable threat intelligence that can be used to enhance security defenses. Analysis results, IOCs, behavioral patterns observed in sandboxes contribute to threat intel feeds used by security teams to improve detection and response capabilities.

2. Cons of Sandbox

a. Evasion techniques

- Some malwares are designed to detect and evade sandbox environments.
- Malware authors employ evasion techniques such as environment checks, delay mechanisms, and anti analysis tricks to bypass detection in sandbox.

b. Limited Realism

- Sandboxes may not fully replicate the complexity and nuances of real-world environments.
- Malware behavior observed in a sandbox may differ from its behavior in a live production environment leading to incomplete or inaccurate analysis results.

c. Resource constraints:

- Sandboxes often operate within resource constrained env, such as virtual machines or cloud instances.
- Limited resources may affect the performance and scalability of sandbox analysis, especially when dealing with large or resource-intensive malware samples.

Other cons of sandboxes include detection overhead and false negatives, complexity and maintenance, Risk of contamination.

Q8. Discuss disassembly and their types/algorithms.

- Disassembly is the process of translating machine code, which is executed by the computer's processor, back into assembly language, This is an essential technique in reverse engineering and malware analysis, as it allows analysts to understand the functionality of compiled programs without having access to source code.

- Types of disassembly:

1. Linear sweep disassembly: also known as sequential assembly, processes the machine code in a linear fashion from the beginning of a code segment to the end.
   - Algorithm - Start at the entry point of the executable. Decode each instruction in sequence until the end of the code segment is reached.

2. Recursive descent disassembly: Control flow analysis starts from an entry point and follows the execution flow of the program, recursively disassembling instructions based on control flow changes such as jumps and calls.
   - Algorithm - Begin at entry point → Disassemble the current ins → If ins is a branch, follow the branch, continue disassembling →

Mark each discovered branch target and repeat the process until all reachable ins are processed.

3. Hybrid Disassembly: Combines the strenghts of linear sweep and recursive descent techniques to provide more accurate and complete disassembly.
   - Algorithm - Start with linear sweep to disassemble large, contiguous blocks of code → Apply recursive descent to follow controlflows and handle branchingmore accurately → Use heuristics and patterns to distinguish code from data and to find additional entry points.

Q9. Explain CPU registers and their types along with examples.

CPU registers are small fast storage locations within a computers CPU used to hold data, addresses and instru temporarily. They play a crucial role in the CPUs ability to process data efficiently. Here's

- Types of CPU registers:

1. General Purpose Registers (GPRs)
   - Used for a variety of operations including arithmeti logic and data manipulation.
   a. EAX (Accumulator) - Often used for arithmetic operations and function return values.
   b. EBX (Base Register) - Used as a pointer to data in mem
   c. ECX (Counter Register) - Used for loop counters and

d. EDX (Data Register) - used for I/O operations and sometimes for arithmetic operations involving large numbers.

2. Segment Registers

- Hold segment addresses; essential for memory segmentation in x86 architecture

a. CS (code segment) - Points to the seg containing current program code.

b. DS (Data segment) - Points to the segment containing the ~~current program code~~ data.

c. SS (Stack Segment) Points to the segment containing the stack.

d. ES, FS, GS : Additional segment registers for extra data segments.

3. Index and Pointer Registers.

- Used for offset addresses in memory operations

a. ESI (Source index) and EDI (Destination index) : used for string operations and memory copying.

b. EBP (Base pointer) : Points to the base of the stack frame, used for stack frame referencing.

c. ESP (Stack pointer) : Points to the top of the stack, used in stack operations.

4. Control Registers :

- used to control various aspects of CPU operations

a. CR0 - Control systems operations
CR2 - contains the page fault linear address.
CR3 - Holds the physical address of the page directory base, used in virtual memory addressing.
CR4 - contains various flags.

5. Status and Flag Registers
Holds flags that indicate the state of the CPU

and the outcome of operations.

a. EFLAGS (x86) - contains flags like zero flags, carry flags, sign flag, overflow flag and others.

b. FLAGS: used to showcase the result of arithmetic and logic operations.

6. Instruction pointer register.

- Holds the address of the next instruction to be executed.

a. EIP (Extended Instruction Pointer) - 32bit x86 architec

7. Floating point register

- Used for floating point arithmetic operations.

a. ST0 to ST7 - The x87 Fpu stack registers

b. xmm0 to xmm15 - SSE registers used for SIMD (Single instruction, Multiple Data)

Q10. Explain 4 types of malwares and their characteristics and behavior.

1. Viruses

- Characteristics

a. Replication - They can replicate themselves by attaching to clean files and spreading when the infected file is executed

b. Activation - They require user interaction to activate such as running an infected program or opening an email with infected attachment.

c. Host dependency - viruses depend on host files to survive and propagate.

d. Payload: Some viruses carry a payload, which could range from annoying messages to destructive actions like data deletion.

- Behavior
a. File infection
b. Boot sector infection
c. Macro viruses
d. Polymorphic & metamorphic Techniques.

2. Worms.
- Characteristics:
a. Self replication. worms can replicate and spread independently without the need for a local host file or human action.
b. Network spreading: They often spread through network connections, exploiting vulnerabilities in network protocols or softwares.
c. Resource consumption: worms can consume system resources, leading to performance degradation or system crashes.
- Behaviour
a. Gmail worms, Internet worms, payload, rapid spread.

3. Trojans
a. Characteristics:
a. Disguise: Trojans disguise themselves as legitimate software to trick user into installing them.
b. No self replication: unlike viruses, trojans do not self replicate.
c. Payload delivery: They often serve as a delivery mechanism for other types of malware, such as ransomware or spyware.
- Behavior:

a. Creates backdoors, RAT trojans, Downloader trojans.

4. Ransomware

- Characteristics:

a. Encryption - It encrypts the infected system's all files or locks the entire system, rendering data inaccessible until the ransom is paid.

b. Monetary demand: Attackers demand payment in exchange for the decryption key or to unlock the system.

c. Social Engineering - Often spread through phishing emails or malicious downloads that trick users into executing the malware.

- Behavior:

a. File encryption, screen locking, Ransom Note, Double extortion, propagation.

Q11. Three functions are commonly found in cases of direct injection: VirtualAllocEx, WriteProcessMemory and CreateRemoteThread. Explain them briefly.

These three functions are Windows API functions used to manipulate the memory and execution flow of processes. These functions are often exploited in malicious activities such as injecting code into other processes.

1. VirtualAllocEx

- Allocates memory in the virtual address space of a specified process.

- Parameters:

a. hProcess: Handle to the process in which to allocate memory.
b. lpAddress: Starting address of the region to allocate (can be NULL to let the system determine its address)
c. dwSize: Size of the memory allocation, in bytes.
d. flAllocationType: Type of memory allocation (eg. 'MEM_COMMIT' to commit memory).
e. flProtect: Memory protection for the region (eg. 'PAGE_EXECUTE_READWRITE')

- Returns: A pointer to the allocated memory, if successful, or NULL, if the action fails.
- Usage in injection: The function is used to allocate memory in the target process where the malicious code or payload will be written.

2. WriteProcessMemory:

- Purpose: Writes data to an area of memory in a specified process.
- Parameters:

a. hProcess - Handle the process whose memory is to be modified
b. lpBaseAddress - Starting address in the target process where data will be written.
c. lpBuffer - Pointer to buffer containing the data to be written.
d. nSize - number of bytes to write.
e. lpNumberOfBytesWritten - Pointer to a variable that recieves the number of bytes actually written.

- Returns: A non-zero value if successful, or zero if the function fails.

- Usage in injection : After allocating memory in the target process, this function is used to write the malicious code or payload into the allocated memory.

3. CreateRemoteThread.

- Creates a thread that runs in the virtual address space of another process.
- Parameters:
  a. hProcess : Handle the process in which the thread will be created.
  b. lpThreadAttributes : Pointer to a 'SECURITY_ATTRIBUTES' structure (usually NULL)
  c. dwStackSize - Initial size of the stack for the new thread (0 for default)
  d. lpStartAddress - Pointer to the application-defined function to be executed by the thread
  e. lpParameter - Pointer to a variable to be passed to the thread function.
  f. dwCreationFlags - Flags that control the creation of thread.
  g. lpThreadID : pointer to a variable that receives the thread identifier.
- Returns: A handle to the new thread if successful, or NULL if function fails.
- Usage in Injection: Function is used to create a new thread in the target process, starting execution at the address, where the malicious code is written. This effectively executes the injected code within the context of the target process.

Q12. Explain the key-logger behavior.

1. Call GetForegroundWindow
- Action - The keylogger retrives the handle of the current foreground window (the window with which the user is currently interacting)
- Logging - If the foreground window has changed since the last check, the key logger logs the new information behavior. This helps identify which application the user is interacting with when the keystrokes are captured.

2. GetAsyncKeyState
- Action - The keylogger iterates through all possible key values (usually 0 to 25) and calls the 'GetAsyncKeyState' function for each key
- Shift and caps lock check: It specifically checks the state of the shift and caps lock keys to accurately log the characters as uppercase or lowercase based on modifier keys.
- Logging - If a new key press is detected, the keylogger logs the key press.

3. Done Iterating through all keys?
- Condition check: The keylogger checks if it has iterated through all possible keys.
- NO - If not all keys have been checked, it moves to the next keys and repeats the 'GetAsyncKey State' check.
- YES - If all keys have been checked, the keylogger loops back to the beginning to call the 'GetForeground

window again and continue monitoring.

4. Loop - The entire process is a continuous loop ensuring that the keylogger constantly monitors the foreground window and captures keystrokes in real time.

Q13)

Aspect	User mode Dbg	Kernel mode Dbg
Scope	Application level	System level (incl OS kernel)
Access	Limited to the address space of the user mode processes	Full access to all system memory and h/w resources.
Safety	Safer as it only targets the targetted application	Riskier, as errors can crash the entire system.
Complexity	Easier to perform and understand.	more complex due to involvement with low-level operations.
Tools	User mode debuggers (eg. VSCode, WinDbg in user mode)	Kernel debuggers (WinDbg in Kernel mode, KD, GDB for linux)
Privilege	Requires standard user level permissions	Requires higher privileges (often admin or root access)
Dbg Env	Can be done on same sys	Requires a separate host.
Breakpnt	Affects the app being debugged.	Can affect the entire system, impacting running process
Interrupt	Interrupts the application being debugged.	Can interrupt entire system causing system wide pauses.

Q14. Explain Reverse engineering process and some important features of IDA Pro.

- Process of reverse engineering:

1. Preparation:
   - Legal considerations - ensure compliance with laws and organizational policies.
   - Tools setup - Install and configure reverse engineering tools like IDA Pro, debuggers and virtual machines.

2. Information Gathering:
   - Collect binaries - Obtain malware samples to be analyzed.
   - Identify Dependencies - Note any libraries or external files the malware might depend on.
   - Gather initial intel - Collect any available info about the malware (eg. from threat intel sources)

3. Static Analysis:
   - Perform basic static analysis of the malware.
   - Disassembly - Use IDA Pro to convert the malware's binary code into assembly language.
   - Code Analysis: Analyze the disassembled code to understand the malware's structure and logic.
   - Data Analysis - Identify data structures, constants and variables used by the malware.
   - Signature matching - Use known signatures to identify common malware libraries and functions

4. Dynamic Analysis-
   - Setup Environment - create a controlled env to

execute the malware.

- Behavior monitoring - use tools like debuggers to observe the malware's behavior during execution.
- Memory Analysis - monitor memory usage and changes to identify injection points and payloads.
- Network Analysis : Observe network communications to understand the malware's external interactions such as command and control server communications.

5. Code Reconstruction:

- Attempt to reconstruct HLL code from the disassembly to better understand the malware's logic and functionality.
- Commenting - Annotate the disassembled code with comments, comments and notes for future reference and easier understanding.

6. Behavior Analysis :

- Determine what harmful actions the malware performs such as data exfiltration, file encryption, file encryption (ransomware) or system manipulation.
- Persistence mechanism : Identify how the malware maintains its presence on the infected system.

7. Modification & Testing:

- Modify the binary to alter its behavior, such as neutralising its payload or disabling its n/w communication.
- Execute the modified malware in a controlled environment to verify that the changes have the

desired effect.

8. Reporting: Prepare detailed documentation of the reverse engineering process, findings and potential mitigations.
   - Share findings with relevant stakeholders, such as incident response teams or security researchers.

- Important Features of IDA Pro:

1. Interactive Disassembly
2. Graphical Representation
3. Multiprocess support
4. Hex-Rays decompiler.
5. High Extensibility.
6. Signature matching
7. Advanced UI and DB Management.

Q15. Write a detailed note on breakpoints and its types.

- Breakpoints are an essential feature in debugging allowing developers and analysts to pause the execution of a program at a specific points to inspect its state and the behavior.
- By using breakpoints, we can gain insights into the program's flow, check the values of variables and understand the conditions leading up to certain events or errors.

Types of breakpoints:

1. Software breakpoints - Implemented by replacing
   - an instruction with a special breakpoint instruction

2. Hardware Breakpoints - Use CPU specific registers to monitor execution without modifying the code

3. Conditional Breakpoints - Pauses execution only when a specified is true.

4. Data Breakpoints: Pauses execution when a specific memory location is accessed or modified.

5. Log breakpoints - Logs information when reached without pausing execution.

Q16. What is hook injection?

Hook injection is a technique used to intercept and modify the behavior of the system functions or the API calls by inserting custom code in a target process. This allows for monitoring, modifying, or redirecting the execution flow of the process

- Working -

1. Identify target function : Determine the function or API call to intercept
2. Inject hook code - Insert custom code into the target process's address space
3. Modify function pointer - Change the function pointer to point to the hook code.
4. Execute the hook code - The hook code runs and performs programmed actions.
5. Return control - Control returns to the original function or is redirected.

- Common techniques

1. Inline hooking - modifies the machine instructions at the beginning of the target function.
2. IAT hooking - Alters entries in the Import Address Table to point to the hook function.
3. EAT hooking - Alters entries in the export address table to ~~point to the hook f~~ intercept calls to exported functions.
4. Usermode Hooking - Hooks functions within the usermode space of an application.
5. Kernel mode hooking - Hooks functions in the Kernel mode space, often used by rootkits.

Q17 Explain 4 sections of the main memory:

Main memory is divided into four primary sections each serving a unique role in program execution

1. Stack

- Stores local variables, function parameters, return addresses, and control information.
- It functions in a last-in, first out (LIFO) manner.
- Automatically managed, memory is allocated and de-allocated as functions are called and returned.
- Fixed size is defined at the start of the program.
- It has a risk of stack overflow, occurs if stack exceeds its fixed size, potentially causing a program crash.

2. Heap:

- Used for dynamic memory allocation, where the size of the data structure can change during program

execution
- Memory is allocated and deallocated manually by the programmer or automatically by garbage collectors in some languages.
- More flexible management than the stack, the heap can grow and shrink as per required.
- It has a risk of fragmentation, overtime, memory allocation and deallocation can lead to fragmentation, reducing performance.

3. Data ~~Text (Code)~~ Segment.
- It stores global and static variables.
- It has two sub-segments:
  a- Initialized Data Segment - stores variables that have initial values set in the program.
  b. Uninitialized Data segment - Stores variables that are declared but not initialized by the program.
- The size is determined by the global and static variables used by the programs.

4. Text (code) Segment.
- Contains the compiled executable instructions of the program.
- It is typically marked as read only to prevent accidental modifications. It contains the binary instructions that the CPU executes.
- It has a fixed size which is determined by the size of the compiled code.

Q 18. Discuss anti-reverse engineering techniques with its types and examples.

Anti-reverse engineering techniques are methods used to protect software from being analyzed, understood or modified by unauthorized parties. These techniques make it difficult for attackers to reverse engineer the software to discover its internal workings, which can prevent piracy, tampering, discovery of vulnerabilities

1. Code Obfuscation
- It transforms the original code into a version that is difficult to understand but still functional. This makes it hard for reverse engineers to read and interpret the code.

eg: Renaming Obfuscation, Control Flow obfuscation, Instruction obfuscation.

2. Anti-Debugging techniques:
- These techniques detect the presence of a debugger and prevent or disrupt debugging activities.

eg. Check for debugger presence, Anti-debugging code patterns.

3. Anti-Disassembly Techniques:
- These techniques prevent disassemblers from correctly interpreting the machine code, making it harder to generate accurate assembly code.

eg. Invalid instructions, code overlapping, jump instruction
- malwares often uses disassembly techniques to hide malicious code from static analysis.

4. Packing and encryption.
- Packing compresses and encrypts the executable code.

making it difficult to analyze without first unpacking or decrypting it.

eg- Packer tools, custom encryption.

- Commercial software protection solutions use packing and encryption to prevent reverse engineering.

5. Code virtualization :

- It translates the original code into a custom intermediate language that runs on a virtual machine making it harder to understand.

eg. VM - creating a vm that interprets bytecode generated from the original code.
VMProtect.

- High security softwares, such as DRM systems, uses code virtualization to protect critical code sections.

Q19. Explain any 9 assembly instructions with proper example and description.

1. MOV (move)

- Transfers data from one location to another.

eg. MOV Ax, Bx ; move value of Bx into Ax

2. ADD (addition)

- Adds two values and stores the result in a register.

eg- ADD AX, BX ; Add value of Bx into Ax

- The value stored in Bx is added to the value stored in Ax, and the result is stored in Ax.

3. SUB (Subtraction)

- Subtracts one value from another and stores the result

inside a register.

eg- SUB AX, BX; Subtract the value of BX from AX.

- The value stored in BX is subtracted from the value stored in AX, result is stored in AX.

4. MUL (multiply)

- multiplies two unsigned values.

eg- MUL BX; Multiply AX by BX

- The value in AX is multiplied by value in BX. The result is stored in AX (lower part) and DX (higher part) if it overflows.

5. DIV (Divide)

- Divides two unsigned value.

eg DIV BX; Divide AX by BX

- AX is divided by BX. Quotient stored in AX and remainder stored in BX.

6. CMP (compare)

- Compares two values by subtracting them, but does not store the result; instead, it sets the flags.

eg CMP AX, BX; compare AX with BX.

- Sets the status flag based on the results of AX - BX (zero flag (1) if equal, carry flag (1) if AX < BX, etc)

7. JMP (Jump)

- Unconditionally jumps to a specified address.

eg- JMP LABEL ; Jump to LABEL

- Execution jumps from the instruction at the addr specified by LABEL

8. CALL (Call Procedure)

- Calls a procedure or function, saving the return address on the stack.

eg. CALL MyProcedure; call the procedure 'myprocedure"

- The address of the next instruction is saved on the stack, and the execution jumps to myprocedure.

9. RET (Return-From Procedure)

- Returns from a procedure, restoring the saved return address from the stack.

eg. RET procedure ; Returns from a procedure.

- The address saved by the CALL instruction is popped from the stack, and execution resumes from that address.