Malware Analysis

REMEMBER

- **Read carefully every question.** Maybe there's something different from the other times.
- We have 3 hours for the exam. It's better to write **the answer to every question** during the analysis, as soon as you get info.
- At the exam we will be given a packed version of the malware and the corresponding unpacked version. After answering the first questions about the packing, (from the 4th on) we can use directly the unpacked version.

Random suggestions

- When a piece of code obviously contains nonsensical commands try right clicking and undefine. It is possible that they are strings interpreted as code.

EXAM QUESTIONS

The first question is about a first analysis

- **PEStudio**: Check strings, sections, resources
    - In the strings check for file names and libraries. Also name things that are semi-readable as **truncated and obsfuscated**
    - If there are two sections and one is writable and executable, other than having a big difference between virtual size and raw size, it most probably has been packed.
    - Check also the resources with **ResourceHacker**: there could be something that is used to load some external code.

The second question is about packing. If you suspect the file has been packed, check also the entropy of the sections (high = packed). Then throw it into Detect it Easy. If it is packed:

- If UPX is detected, scan with UPX (upx -l "filename")
- Search for the **tail jump**
    - Check for all the jumps in the code (a simple search for jmp should do the work, but possibly a jnz, jz.. could be necessary
    - Search for a long jump (to another section)
    - Check if the code it points to seems not to have any sense (it's packed)
- Screenshot everything and then delete the imports that generate errors. At this point dump everything into a new file and then click on fix dump. The new file will have the whole IAT rebuilt this time. Remember to check if the new file is compatible with the one unpacked provided by the prof. It can be done with an analysis of the imports of the two in PE studio.

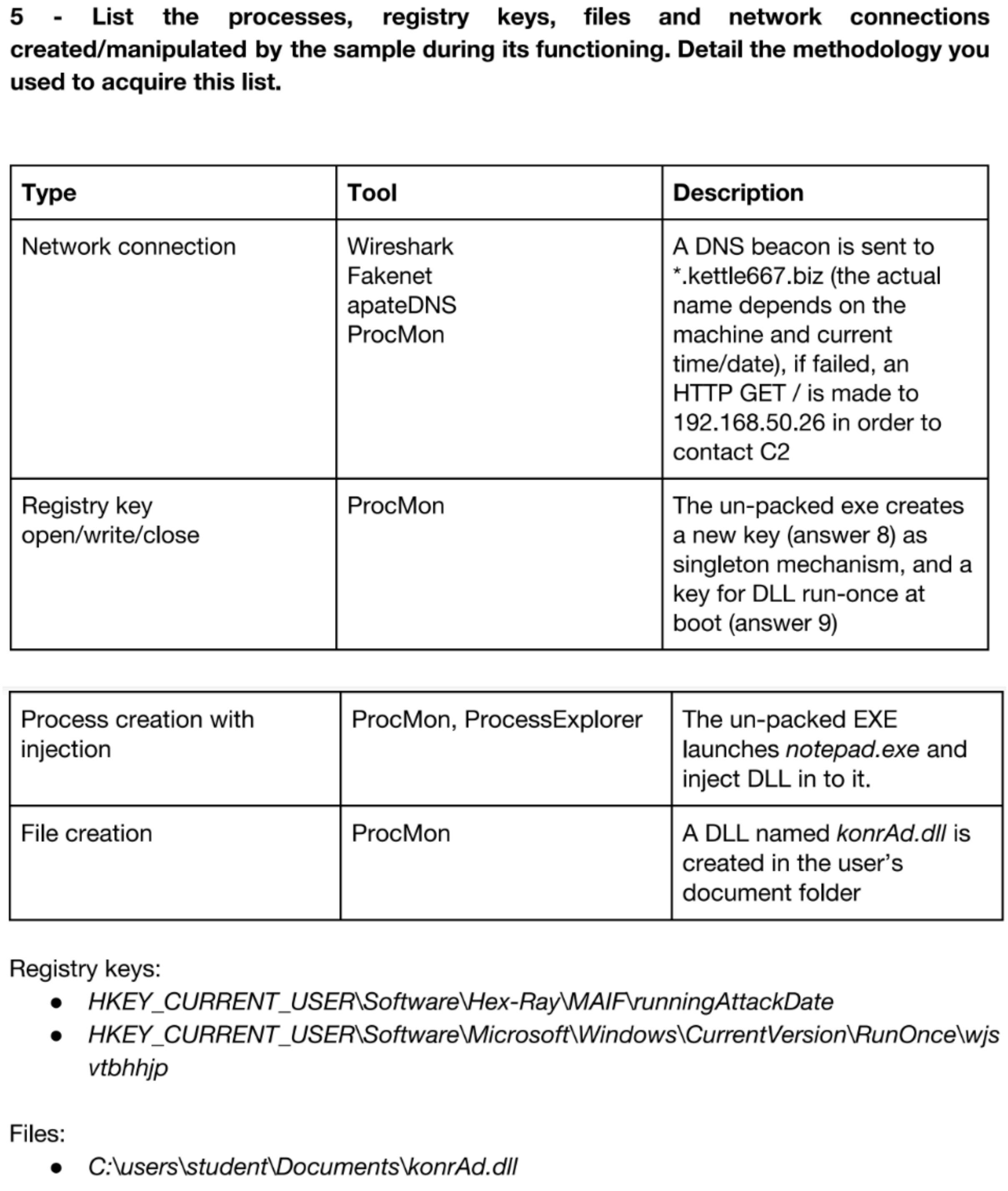
The third question asks to reconstruct the IAT with the original OEP. Lock the program in the debugger at the address it was tail jumped to, and put the address on Scylla to find the IAT.

- Sometimes the jump found this way is not the only tail jump. If you can't reconstruct the IAT at this address try to find another jump inside the code of the new section.
    - In particular if it is packed with MPRESS and you have trouble finding the OEP you can try to find an instruction "popa". It is the instruction that terminates the demangling procedure and is often followed by another jump that leads to the OEP.
- To check if two unpacked files are equal go to imports in ida and screenshot that they have the same imports

The fourth question is a high level description of the malware. Should be very short as the specific parts are explained better in other answers. It's better if left for the very end.

The fifth question is a list of processes, registry keys, files, network connections. This is also better left for the end. To see which files/processes/registers are modified use Process Monitor on the unpacked version.

- Go to registry editor and show the freshly created registry. Not the ones found using ProcMon (too many) but **the ones you found during static analysis**.



5 - List the processes, registry keys, files and network connections created/manipulated by the sample during its functioning. Detail the methodology you used to acquire this list.

Type	**Tool**	**Description**
Network connection	Wireshark Fakenet apateDNS ProcMon	A DNS beacon is sent to *.kettle667.biz (the actual name depends on the machine and current time/date), if failed, an HTTP GET / is made to 192.168.50.26 in order to contact C2
Registry key open/write/close	ProcMon	The un-packed exe creates a new key (answer 8) as singleton mechanism, and a key for DLL run-once at boot (answer 9)
Process creation with injection	ProcMon, ProcessExplorer	The un-packed EXE launches *notepad.exe* and inject DLL in to it.
File creation	ProcMon	A DLL named *konrAd.dll* is created in the user's document folder

Registry keys:

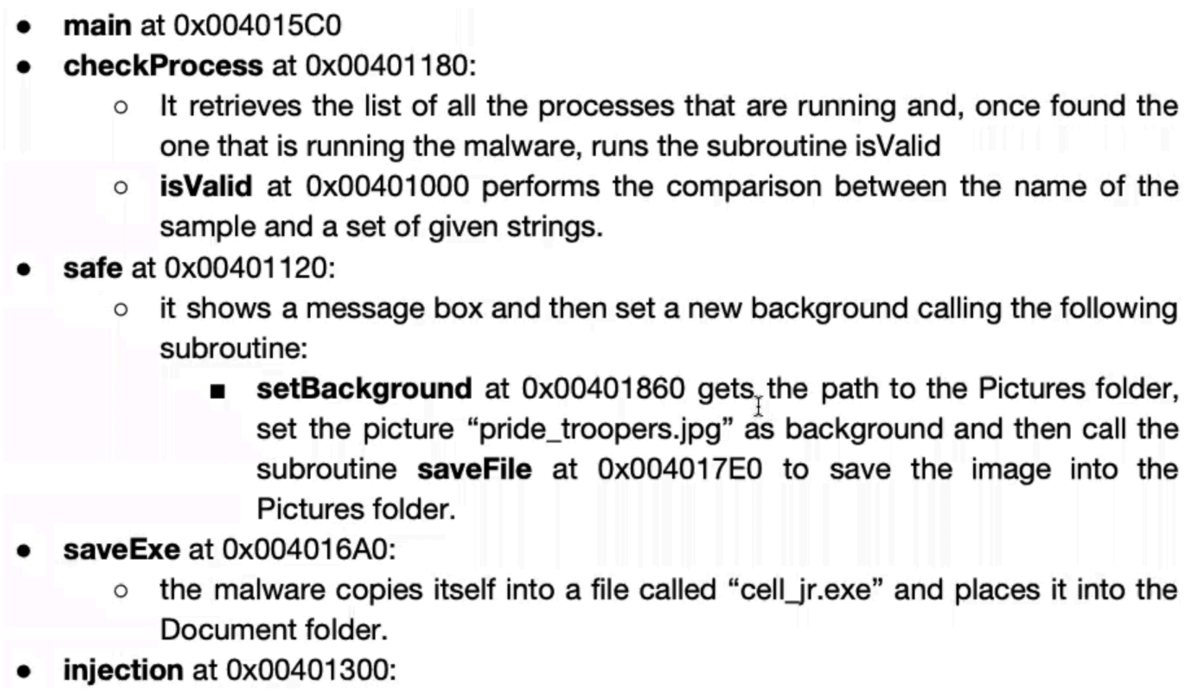
- *HKEY_CURRENT_USER\Software\Hex-Ray\MAIF\runningAttackDate*
- *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce\wjsvtbhhjp*

Files:

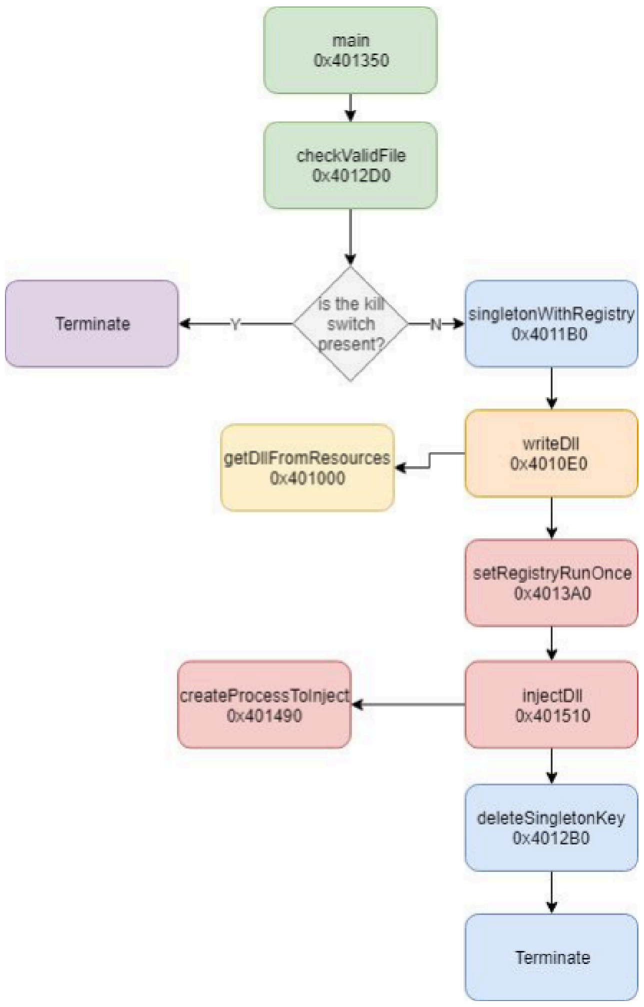
- *C:\users\student\Documents\konrAd.dll*

This is a good example of answer to question 5

The sixth question asks about subroutines (functions). Write addresses for each subroutine and explain when and how it is used. Give also to each function a name. It's basically an advanced analysis with ida. **Remember to decompile with IDA also the dll file if a dll injection is performed, and to list and analyze its functions.**



- **main** at 0x004015C0
- **checkProcess** at 0x00401180:
    - It retrieves the list of all the processes that are running and, once found the one that is running the malware, runs the subroutine isValid
    - **isValid** at 0x00401000 performs the comparison between the name of the sample and a set of given strings.
- **safe** at 0x00401120:
    - it shows a message box and then set a new background calling the following subroutine:
        - **setBackground** at 0x00401860 gets the path to the Pictures folder, set the picture "pride_troopers.jpg" as background and then call the subroutine **saveFile** at 0x004017E0 to save the image into the Pictures folder.
- **saveExe** at 0x004016A0:
    - the malware copies itself into a file called "cell_jr.exe" and places it into the Document folder.
- **injection** at 0x00401300:



This is an example of a good answer. Use diagrams.net to create the schema.

The seventh question asks about queries on the surrounding environment. Therefore here should be written things like killswitchs based on time, geographical location, content of the filesystem.

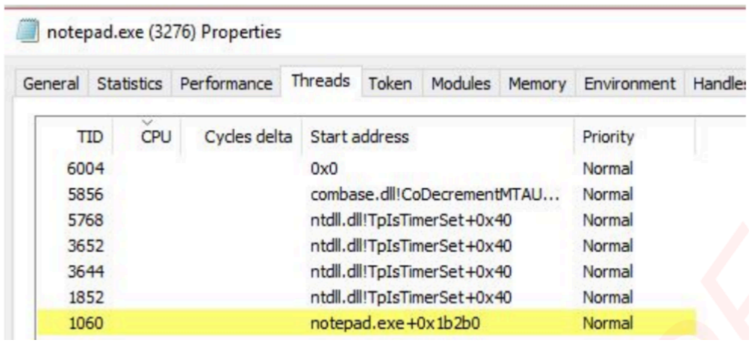
- The combination of K32EnumProcessModules and K32GetModuleBaseNameA should be a red flag in that sense. It is most probably searching for some specific process with a substring to decide if continue or not with the execution of the malware.
- Also, the presence of a mutex can be interesting. It's a mechanism to understand if there is only one version of the program executing.
- Check for "singleton" mechanisms: the malware could be searching if there are other copies of itself running.

- see if there is a check in your code that checks if the process is already running. **If there is:** write in the writeup that a simple test (see below) can confirm this hypothesis.
- If you don't see any singleton just try to run the program twice: enter the program in debug mode from ida and hang it at a certain point, and then run the program again from the command line.

The eighth question asks about persistence mechanisms.

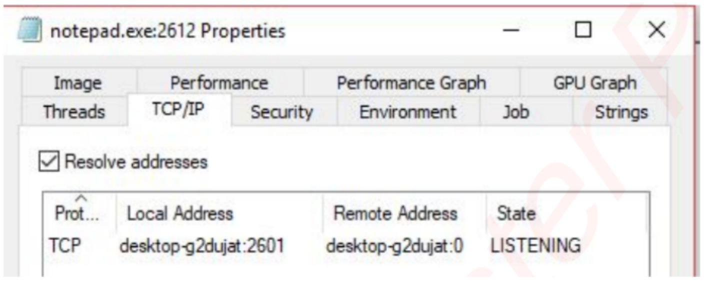
- In this category can be inserted things like: writing of the malware path in the registry keys, copy of the malware in the startup folder.
  - Registry Keys: The malware creates registry keys in the Windows registry to ensure that it loads every time the computer starts.
  - Startup Folder: The malware adds itself to the Startup Folder, which runs automatically when the user logs in.
  - Services: The malware creates a Windows service, which runs automatically in the background, even if the user is not logged in.
- Check also for the method SetFileAttributesA, as it is often used to make a file read-only (to make it more difficult to eliminate)

The ninth question is about code injection. An injection can be (and actually most of the time is) performed on processes created by the malware itself. It's just a way to disguise bad actions making fake normal processes do them.
REMEMBER: Check with process hacker after launching the malware (try both with and without breakpoints) if the process or the shellcode is doing something, also web activity. This is also **super useful to analyze behaviour of shellcode.**


notepad.exe (3276) Properties

General | Statistics | Performance | Threads | Token | Modules | Memory | Environment | Handle

TID	CPU	Cycles delta	Start address	Priority
6004			0x0	Normal
5856			combase.dll!CoDecrementMTAU...	Normal
5768			ntdll.dll!TpIsTimerSet+0x40	Normal
3652			ntdll.dll!TpIsTimerSet+0x40	Normal
3644			ntdll.dll!TpIsTimerSet+0x40	Normal
1852			ntdll.dll!TpIsTimerSet+0x40	Normal
1060			notepad.exe+0x1b2b0	Normal


notepad.exe:2612 Properties

Prot...	Local Address	Remote Address	State
TCP	desktop-g2dujat:2601	desktop-g2dujat:0	LISTENING

It can be:

- **DLL injection:** it has to be analyzed in IDa
- **Shellcode injection** (look at the WriteProcessMemory function): the behavior of the shellcode is not necessary for the mark but can be used for the laude. **Remember** to say that its a shellcode injection if there is **VirtualAlloc** to allocate memory on the victim process and uses **WriteProcessMemory** to write the payload into the process.

  Blobrunner can be used for this: find the memory address containing the shellcode by putting a breakpoint in the program when it is about to inject the payload and analyzing the registers. Then parse the hexadecimal, select the lines you are interested in, export them to a.bin file and open them with blobRunner. (**Better alternative**: find in the registers before the WriteProcessMemory the lpBaseAddress. Then open ProcessHacker, choose the infected process, double click and go to memory. That's the shellcode, you can save it). Then open the blobrunner executable in ida and debug the blobrunner process again. Navigate to the entry address which is provided in the terminal by blobRunner. If necessary with a right click convert everything to code, or **put a new breakpoint** in the first instruction and continue the execution, to make the code appear. At this point the shellcode should be visible.

  When parsing shellcode you may find functions hashed so that they are not readable. Most of the time by searching online you can trace the original function.

The tenth question is on network behavior. Per fare Network Analysis avvia questi programmi: Wireshark, Fakenet, apateDNS. Poi runna il programma e vedi cosa compare

The eleventh question is about obfuscation actions performed in the sample. Obfuscation techniques can be used in various ways, such as: ù

- string obfuscation/encoding (**Note when something is moved char by char**)
- dynamic loading of multiple libraries at runtime (**Remember to screenshot them!**)
- **Xor or Base64 encoding.**
- Also using of resources is considered as obfuscation
- Also, loading functions at runtime that are never used.