1 Basic static techniques

Static analysis = process of analyzing the code or structure of a program to determine its function. The program is **not** executed.

Main techniques

1. Using antivirus tools to confirm maliciousness

AV tools identify suspected files by relying on a database of: - identifiable pieces of known suspicious code (known as *file signatures*) - behavioral and pattern-matching analysis (*heuristics*)

Problems of this approach: - Only known malware can be identified. - Malware's code can be modified to change the signature and try to evade virus scanners.

Suggested tool: VirusTotal, allows to scan using multiple antivirus engines.

2. Using hashes to identify malware

Hashing is commonly used to uniquely identify malware. Many tools (also VirusTotal) allows to search for an hash and check if it has already been identified as malicious.

Suggested tool: `sha1sum` and `md5sum` on Linux, to determine sha-1 and md5 digest of a given file.

3. Gathering info from file's strings

Searching through strings can be a simple way to get hints about how the program works. 2 ways to store strings: - ASCII: 1 byte per char, terminate with NULL - Unicode: 2 bytes per char, terminate with NULL

ASCII

B	A	D	NULL
42	41	44	00

ASCII representation of the string *'BAD'*: the string is stored as the following bytes `0x42, 0x41, 0x44, 0x00`.

UNICODE

B		A		D		NULL	
42	00	41	00	44	00	00	00

UNICODE representation of the string *'BAD'*: the string is stored as the following bytes `0x42, 0x00, 0x41, 0x00 ,0x44, 0x00 , 0x00, 0x00`.

Suggested tool: `strings` on Linux, to search an executable for a 3-letter (or greater) ASCII and Unicode sequence followed by a string termination char. Some detected strings are not actual strings; ex. `0x56, 0x50, 0x33, 0x00` can be detected as *'VP3'*, but those bytes could be a memory address, CPU instructions or something else. The user should be able to filter the invalid strings.

Packed and Obfuscated Malware

- **Obfuscated**: programs whose execution the malware author has attempted to hide.
- **Packed**: subset of obfuscated, where the malicious code is compressed and cannot be analyzed.

NOTE: Packed and Obfuscated code will include at least `LoadLibrary` and `GetProcAddress` to load and gain access to additional functions.

Packing files

When a packed program is executed, a small function runs to decompress the packed file and then run the unpacked file. If analyzed statically, only that small function would be dissected (the rest is hided behind compression), thus it needs to be unpacked in order to be analyzed.

Packed files can be detected with EXEInfo PE.

Suggested tool: UPX allows to unpack program simply using `upx -d PackedProgram`.

PE File Format

Portable Executable (PE) file format is used by Windows executable, object code and DLLs; it provides information to the Windows OS loader to manage the wrapped executable code. For this reason, it is used by almost every file with executable code loaded by Windows. The header of PE files contains info about the code, type of application, library functions, and others.

Linked libraries and functions

Imported functions could be useful to understand the behavior of an executable. 3 types of linking libraries:

- **Static linking**: (common in UNIX), all code from the library is copied into the executable. This may make it difficult to understand what is the executable's own code, since the PE file header do not indicate if the file contains linked code.

- **Runtime linking**: (common in packed and obfuscated malware), executables connect to libraries only when a specific function is needed.
- **Dynamic linking**: executables connect to libraries at the execution of the program. PE file header stores information about loaded libraries and functions called by the program.

In Windows, some functions allows to import linked functions not listed in the program's file header. Some examples are `LoadLibrary` and `GetProcAddress`, which allows to access any function in any library on the system; thus when these are used it is impossible to detect with static analysis which functions are being linked by the suspected program.

Exported functions

Typically DLL files export functions to make them available to executables; the PE contains info about which functions are exported.

Some interesting Windows findings when exploring imports

- `SetWindowsHookEx`: receive keyboard inputs (used by keyloggers)
- `RegisterHotKey`: receive hotkeys (used by keyloggers)
- `Advapi32.dll`: registry is being used (search for registry key strings)

Investigate PE

File headers

Interesting sections in a PE file: - *.text*: contains instructions that the CPU executes - *.rdata*: import and export info (sometimes in *.idata* and *.edata*) and read-only data used by the program - *.data*: program's global data - *.rsrc*: resources used by the executable (such as icons, images, menus, strings..)

PE files

- *IMAGE_FILE_HEADER* entry contains the timestamp when the executable was compiled.
- *IMAGE_OPTIONAL_HEADER* entry tells us if the program has a GUI (*IMAGE_SUBSYSTEM_WINDOWS_GUI*) or not (*IMAGE_SUBSYSTEM_WINDOWS_CUI*).
- *IMAGE_SECTION_HEADER* entry contains info about each section of a PE file, and allows to detect packed executables (if *.text Virtual Size* is much larger than *Raw Data*).

Resource section

Collection of the icons, menus, dialogs, strings, version info.. Similar to resources for Android apps.