A.M. Anisul Huq
82372J

Report: Assignment on Special Course in Information Security – Malware Analysis (2010)

Description of the architecture:

The tool which I have implemented for this assignment was created after examining how the code in the test files has been obfuscated. Observing that after the entry point of each of the provided files there is a loop, where the values stored at a certain offset in the .text section are XORed with a value of which the initial "key" is provided in the BL register, I deduced that the deobfuscation of the code was done in practice by that simple XOR decryption. This occur before a JMP instruction, which moves the execution to a totally different offset (far before the EP of the file), towards a code set in a way to typically raise some kind of exception, preventing the computation to end successfully, the program being instead terminated.

Starting from these hypothesis, I thought about a simple way to retrieve the actual obfuscated data, and detect whether there where any detectable differences in the two sets provided, if not even among each set of detectable and non-detectable files. To do this, the code immediately following the EP is scanned until the mentioned "suspicious" JMP instruction (which might be an indicator of using an obfuscation technique, as in fact here the code after the EP is solely necessary to this purpose) is reached. At this stage, the "initial decryption key" (the hex value initially stored in the BL register) to start develop the obfuscation layer on the data, as well as the boundaries (offsets) in the file where these data are contained, are saved by the script. The data is then decrypted automatically by our tool, reproducing the behavior of the file itself as it removes the obfuscation layer, and the data is returned into a string of hex values, which we can say represents a sort of "signature" of the file.

At this stage, when one of the files that should be detected as malware is analyzed through this process, the result (its "signature") is stored as a "template" for the supposedly harmful files – i.e. it becomes a known malware sample's signature - and each of the following analyzed and deobfuscated files is compared – in that returned piece of decrypted data – to this template. Based on this, the tool identifies as malware all the files which have the same signature of the chosen known sample (in this case "detect1.exe", but it might have been any of the "detect" executables), while judge as harmless all the files which have a different signature.

Pros & Cons, Future work and improvements:

It comes straightforward that this decision is not generic, and has apparently no real-life application out of the scope of this assignment, but I made so that the tool is highly extensible and customizable to incorporate new features that could detect other kind of malware. For example, an option is provided to analyze the signature of each file, using a text file which is a database of know signatures from PEiD, thanks to the module Peutils. It would be even trivially possible to save the signature of the known harmful samples, adding it to the existent database file, or creating a new database for future found signatures, so that when the tool encounter the same behavior more than once, it's able to recognize it as malware. Another option for a dump of the "signatures" of each file is provided in the tool. These two ideas were not totally integrated into the script I developed, simply due to the fact that the other features mentioned above were enough to detect the correct behavior of the test files provided. These are part of the architecture, inserted for a future, more generic, improvement of the script.

Other ideas left for future development were methods to detect if the file is packed (judged as useless here, since all of them had the same characteristics in this sense), analyzing the entropy of the sections, the number of imports, section names, Windows API calls, and so forth. Lastly another idea was to re-write (when, and if possible, since in this case it seemed possible but it's more likely that usually malware have no writing rights) the obfuscated data in the file, and substituting them with the correspondent deobfuscation, eventually modify the EP in consequence, to verify the actual data, thus creating a “normalizer” for the code. This might be however expensive in terms of computational effort and resource usage, particularly in case of longer files, or an high number of samples. Without mentioning that there still might be multiple obfuscation layer, and these haven't been considered in the tool.
A good aspect of this tool is the ability to scan through the files really fast, in the order of circa a couple of seconds for the whole ten samples. This is due to the fact that only the instructions immediately after the EP are scanned, and the other “predominant” loop in the implemented code is the removal of the obfuscation layer, which in our case is quite trivial and relatively short. Only the signature of the known detectable sample is globally stored, thus trying not to consume memory unnecessarly.
As mentioned, the solution is not generic, but can be easily extended and customized. For example if a different file with different charactheristics would be scanned by this deobfuscating engine, it would simply be recognized as harmless (at a first attempt I though it would have been useful to recognize it as a third, different sample, comparing it also to the “dontdetect” files, but that was out of the purpose of this analysis), since it's different from the “detect” samples. On the other hand, if an hypothetic “detect6.exe”, with the same behavior of the other five, was introduced in the set, it would be recognized as malware.
Hence, is not taken into account a different obfuscation method, and even with the same obfuscation, if the actual code resulting from the deobfuscation is different, it wouldn't be threated as malware, leading obviously to a huge number of false negatives. Due to the customized nature of the tool, however, there couldn't be false positives, since the decision is made in relation to that particular signature, unless a proper signature database is provided and the “-s” option enabled. This option could potentially allow to recognize a set of known malwares that match the signatures saved into the given database, but I wasn't able to test it in practice due to the difficulty in retriving samples of those particular malware. In this case the speed in the computation would be significantly sacrificed due to the scanning through the whole database for each sample (this could be improved by making so that only the suspicious signatures retrieved from the deobfuscation are later compared with the signature database).
This “signature-based” tool, as mentioned, provide also a dump of each file's deobfuscation, which might be find useful for a low amount of data (e.g. in this case), but the idea would be to adapt this option to provide in the dump only the malware's signatures (or anyway the “suspicious” signatures), in order to save time and resources in the analysis of a huge amount of executables. It's obvious that since the script uses the Pefile module, its application is restricted to this particular kind of executable files.
An alternative way of providing a more generic solution, which I have considered to take into account, would be to handle in the code the detection of the most common obfuscation methods, so that the obfuscation layer could be removed (and possibly the resulting code re-iterated to detect multiple layers), and the actual code be analyzed.
All the improvement described represents possible changes and additions for making the tool more generic and improve its performance, making it able to be used against real world threats, starting from a simple, signature based approach, and eventually enhancing it further to analyze more complex malware with different obfuscation techniques and multiple obfuscation layers.

Performance evaluation:

In terms of performance, the tool is able to analyze the ten samples provided (and correctly detect

the two different sets of five files each) in an amount of time between 0.2 and 0.3 seconds on average. This makes it suitable to realistically analyze in this trivial way a wide number of samples, independently from their nature. On the other hand, the tests done enabling the optional checking of a possible match with the signatures in the PEiD database, slowed down the computation in the order of about 6.6-6.8 seconds, with a degradation in performance of approximately 30 times, for only 10 samples. The implications show obviously that, in case this function should be applied in the real world, monitoring thousands of files, it would be strongly suggested to apply some modifications such as what I've mentioned before in the previous section of this report.
The same problem is present in case of resource consumption and disk usage. In its simplest case, the tool would probably handle the analysis without creating any significant overload on the system (and no impact on disk usage). Realistically speaking, it's not feasible to think to analyze thousands of files, and just print the result of each in the output (in this implementation I've chosen to print out also the result for the “dontdetect” files, only in order to clarify that even these are analyzed by the tool, otherwise only the files considered as malware would have been signaled). The consequence of this would be to print the results on a file, if the output is likely to be big enough. Moreover, in the signature matching approach, several numbers of additional reads to file and comparisons would be performed, and I've already showed how this would impact the performance in terms of time, directly related to a proportional increase in computation and operations performed by the system. The disk usage might be taken in consideration in two main cases: the creation of a signature database from the suspicious malware signatures, and the dump of a large sample of files.
Both could be significantly big in case of a “massive” sample analysis, especially the second case. But realistically in the worst case scenario, as the signature so far is relatively small, the impact on disk shouldn't be bigger than some Mbytes, thus not being relevant at all for nowadays hard disks.
The nature of the tool makes so that all the samples analyzed are threated in the same way, so that, as it's implemented at the moment, an essentially constant execution time would be taken for each sample, making the performance degrade linearly in proportion with the number of analyzed executables. If different methods, such as what I've discussed previously about writing signatures only for suspicious files, adopting different approaches for different obfuscation methods, or further scanning to verify the existence of multiple obfuscation layers, would obviously cause a variable execution time for the analysis of each sample, affecting the performance in a more complex way, which is not easily predictable without taking into account the various possible paths.

Attempting a more specific evaluation of the execution time and space consumption by the current implementation of the tool, a rough estimate lead to the following results:

$T = O(1) + O(n) + O(m^2) = c.ca\ O(nm^2)$

for the function “deobfuscate(filename)”, where O(n) is due to the scanning after the entry point, and “n” is the number of instructions analyzed, and $O(m^2)$ is due instead to the deobfuscation, where “m” is the number of operations performed on the data which are iteratively XORed.
The time of the “main()” function is so calculated by:

$T = O(n) + O(nm^2) = O(nm^2)$

Which is the total estimated time. This result is however dependent from unknown times from the functions loaded from external libraries, and used to parse and disassemble the code, as well as optionally (enabling the signature matching) with the comparisons with a signature database of theoretically variable length.

The memory space used is instead roughly estimated in:

$S = O(m)$

Where “m” is the length of the data which is analyzed.

Dealing with false positives:

As said before, the tool wouldn't be prone to false positives, due to the specific custom nature of the actual implementation. It would be instead very likely to give false negatives, unless the malware samples has the same obfuscation method and the same deobfuscated code of the detectable executables provided for this assignment. For this reason the tool is not considered to be ready for a realistical application “on the real world”.