Report on Anti-Malware Engine

Vikash Jha

Introduction

The task in hand is to develop an anti-malware engine that detects a malicious code. The engine has two parts namely the de-obfuscator and the malware checker. As the name suggests the de-obfuscator removes the layer of obfuscation of the code and malware checker checks for the malicious code. These components achieve this by emulating the code and running a finger print scan and memory scan on binary images of the PE file. Below sections covers the algorithm and tools used in greater detail.

Tools

The Anti-virus engine uses pyEmu, pydasm and pefile libraries.

- **pyEmu** : used to emulate the code. It is very helpful in running the code from its entry point to the point till the control transfer to the real code .Execution of the code in the emulator helps in de-obfuscation of the real code.
- **pefile**: Used to parse the PE file. It also helps in retrieving a section of image from the PE file.
- **Pydasm**: It is a dissembler and this helps in dissembling the binary image into mnemonic instructions.

Architecture

I have implemented a class 'AVEngine' that implements methods to check the obfuscation and to detect malware. The class checks the obfuscation using a fingerprint and then it check for maliciousness of the program using another fingerprint along with memory scanning. Various methods and their functionalities are described below:

- **Init** : Initialize the engine
- **Isfileobsfucated**: Checks if the file is obfuscated
- **Checkmalware**: Check if is a malware
- **Deobsfucate**: De-obfuscate the code
- **SearchObsSig** : Check for presences of obfuscation's signature
- **SearchCodeSig**: Check for presences of code signature

The various component of the engine is shown below:

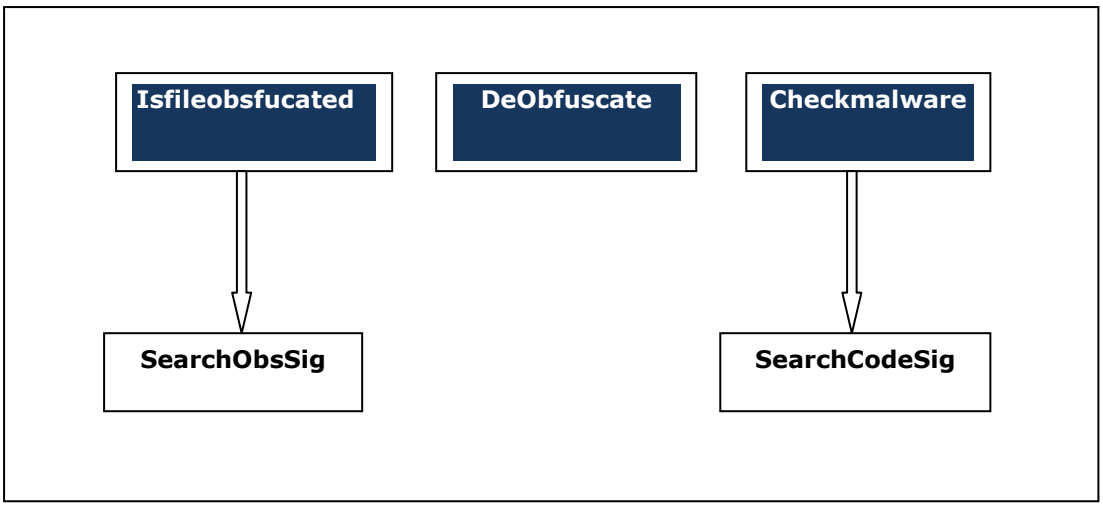


Figure -1 Components of Anti-Malware Engine

We will go through the design considerations and algorithms of the methods below.

Isfileobsfucated :

It takes 40 bytes from the entry point of the PE file and scans for the following fingerprint . The scanning supports wild card search

0xB9*0xBE*0xB3*0x8A0x160x320xD30x8A0x1E0x880x160x460xE2*0xE9

'*' denotes one or more character followed after the preceding character and before the next character

If the finger print is found then it indicates that file is obfuscated using the method of XORing the byte with the previous byte and we have a key with which the first byte is obfuscated. The wild character search gives the flexibility to match the signature for other variants of same obfuscators.

Checkmalware:

This method scans for the presence of the fingerprint "0x68***0x6A0x010x6A0x000xE8*0x68 *0xE8*0x68*0xE8**" in the actual user code('*' denotes one or more character followed after the preceding character and before the next character), if the scanning is successful, then it checks for the presence of string "**you must detect this file**" in the memory . If both the search is successful then it flags code as malicious .Algorithm used in this method can be outlined below

```
        If  file is obfuscated
                Copy no of obfuscated bytes from the address in EIP register.

        else
                Copy 100  bytes from the entry point address
          If scan of fingerprint and string "you must detect this file" is  successful   in
copied
                bytes
                  Flag as Malware
        else
                Healthy File
```

SearchObsSig:

This method searches for **obfuscation finger print** in the given text. This search supports a wild card '*' meaning any number of character followed after the preceding character and before the next character in pattern will also be taken into account. Wild cards add a flexibility to modify the fingerprint at a later point of time to accommodate slight variants of same malware. Algorithm used in this method can be outlined below.

```
Split the patterns on the wild card character '*' into pattern
For each pattern
      Find pattern in the text
      If not successful
            return unsuccessful
     reduce text to start from the next character just after previous successful search
return unsuccessful
```

SearchCodeSig:

This method searches for **code finger print** and string "You must detect this file" in the memory locations. This search supports a wild card '*' means any number of character followed after the preceding character and before the next character in pattern will also be taken into account. Wild cards add a flexibility to modify the fingerprint at a later point of time to accommodate slight variants of same malware. We search for string in that memory location which appears as immediate constant in the code. This helps us to find the string faster as we do not have to scan the entire memory. Algorithm used in this method can be outlined below.

```
Split the patterns on the wild card character '*' into pattern
For each pattern
      Find pattern in the text
      If not successful
            return unsuccessful
      if the instruction has immediate constant as memory address and stringfound == 0
            scan the memory address for the presence of string "you must detect this file"
             set stringfound to true
     reduce text to start from the next character just after previous successful search
return unsuccessful
```

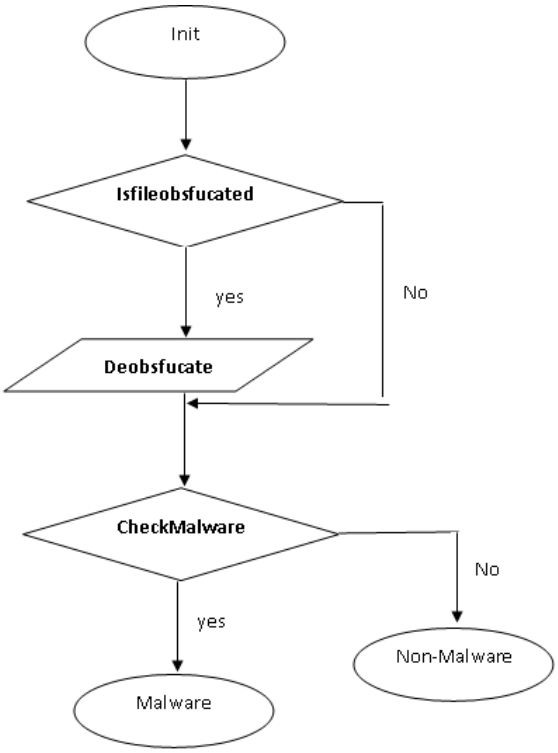
Deobsfucate:

This method de-obfuscates the real code by running the de-obfuscation code in emulator . This method is tightly bound to the method of obfuscation . If we want to extend the library to handle different obfuscators then we have to write different de-obfuscate module for those .For current obfuscator the algorithm for de-obfuscation is below

```
Set ECX,BL and DL to 0
While not ECX and not  BL and not  DL
      Execute the instruction
While the value of ECX not zero
       Execute the instruction
Execute the jump instruction
While instruction not jump
      Execute the instruction
EIP points to the real code
```

Application of Library

AVEngine is a class so it can be called by any application to determine whether the file is malicious .The application that uses this library should call **Isfileobsfucated** to check whether the PE file under observation is obfuscated .If the PE file is obfuscated then it should call **Deobsfucate** to remove the layer the obfuscation . Then it should call the **Checkmalware** to check whether the file is malicious or not. In flow chart it can be depicted as follows:



flow chart showing api call

False Positives

I will analyze this section first with respect to the assignment and then slightly in generic sense. We have been given 5 good and bad files in the assignment. When we look closely all these files are obfuscated with the same obfuscation logic (XORing the bytes with the previous bytes, with a key for the first byte) and performs the same function of printing a string on the terminal. When we observer carefully, we find that bad files creates a mutex and prints "You must detect this file", where as the good file does not create a mutex and it just prints "You must NOT detect this file". So while checking for maliciousness of the code we check for the finger print that takes care of create mutex part and apart from this finger print, we also scan the memory area for the presence of the string "You must detect this file". I think scanning for these two may act as a good and sufficient differentiator for good and bad files. I tried testing with the slight variants of the good and bad files apart from the one given in the assignment (e.g. remove the obfuscation part from the bad file then also it should detect the file etc). The library performed correctly.

If we look at this problem slightly in a generic sense we have to cover all the case where a good file can be similar to bad file and design our heuristic in such a way that it should take all factors into consideration. The implementation provides a flexible framework that can be extended to incorporate other heuristics.

Advantages

The design approach discussed above has some advantages which are listed below.

i) There are two fingerprints used in the engine, one for checking the obfuscation other for the checking the code. These two signature scanning mechanism is kept separate and this will help in extending the library for different obfuscators and malwares independently as it will just require adding the new signature for the component that will be checked.

ii) As the entire class is laid out in a generic sense and is independent of the application so this can be used as a separate library .It can also be extended for various other functionalities e.g. decompression module can be added easily.

iii) The scanning mechanism supports wild card search, this will help in giving flexibility to the fingerprints search . Some small variants of the signature can be added without modifying the engine.

iv) As the library supports the memory scanning of the relevant string ("you must detect this file" in this case) this will help in removing false positives as there can be a many files which will have similar code fingerprints but may not print this on screen. With the memory scanning the library in a sense tries to figure out the functionality of the PE file under observation

Improvements and Future work

As currently works with the files of the complexity of "hello world" so there a lot of scope of improvement in the library. Some of them are listed below:

i) The library can be extended to include, decompression functionality for the PE files.

ii) The de-obfuscation logic can be extended to work with the obfuscators of wide range

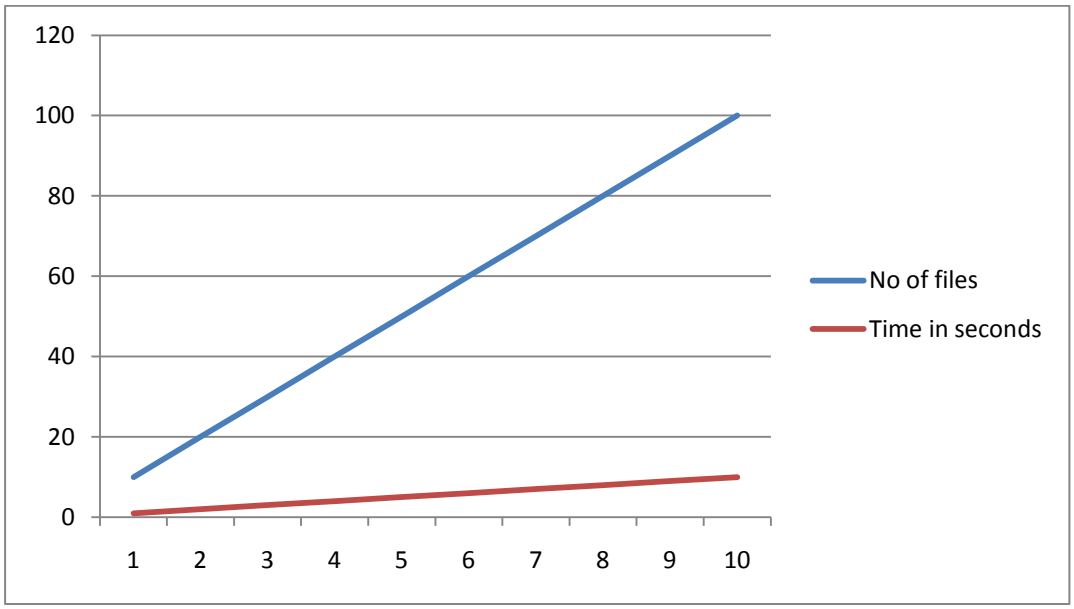
iii) More amount of heuristic apart from memory scanning should be added to cover the wide range of checks and balances to cover all types of malware

iv) Lastly, the library should be tested with a wide range of good files to really understand how the logic behaves with the false positives. I tried testing with a 3-4 different variants(apart from the given) of "hello world " program it behaved correctly but a lot of testing needs to be done to really verify a logic against false positives.

v) Wild card search can be extended to include various other cards.

Performance

The major functionality of the library is of linear complexity, so we can say that the efficiency of library in detecting the malware is linear i.e. O(n) where n being the number of file . As far as disk usage is considered the library does not use any additional storage in disk apart from the files that are analyzed. The library is also not very extensive in memory usage as usages of the variables are optimized.



Performance graph

Conclusion

The class AVEngine successfully detects the malicious files given for the assignment using fingerprint and memory scanning. The performance of the library is linear and it does

provide a very flexible architecture that can be extended. A lot of work needs to be done to extend this class for real world usage.