# Unpacking Python Executables on Windows and Linux

fortinet.com/blog/threat-research/unpacking-python-executables-windows-linux

May 3, 2022

Threat Research

By Gergely Revay | May 03, 2022

Traditional programs written in the python programming language are distributed as source code and the python interpreter is used to run them. This is easy if one runs their own python code; however, it is rather cumbersome to deliver commercial products this way. To help with that, a couple of projects were created that can bundle a python program with all its dependencies into an executable file: Portable Executable (PE) on Windows and Executable and Linkable Format (ELF) on Linux/Unix.

Python malware is also distributed as such packed executables. And if we talk about malware, the question always come up, “how can we unpack and decompile the malware to look at its python source code?” I discussed this topic in a video I created two years ago. But since then, new python versions have come out and the unpacking techniques have changed.

In this blog post, we are going to go through the following topics:

- Packing
- Unpacking and decompiling on Windows below python version 3.9
- Unpacking and decompiling on Linux after python version 3.9

Differentiating between the older and newer python versions is important since a lot changed after python 3.9, both in how python bytecode is generated and how (and whether) the source code can be recovered.

**Affected Platforms:** Windows, Linux/Unix

## Packing

First of all, let's discuss what python packaging is—and specifically, PyInstaller. (Note: I use the terms packaging, packing, and bundling interchangeably.) The goal of packaging a python program is to create an executable that can run independently on an operating system. We should not confuse this with general malware packing, where the goal is to hide malicious code from analysts and security tools. Python packaging does not intend to provide any security or obfuscation. It is only a side-effect of the packaging. When we bundle a python program, the tool that we use for packaging, such as PyInstaller, does the following:

- Compiles all .py source files to python bytecode (.pyc files)
- Collects all python compiled source code and python dependencies
- Includes the operating system-dependent python interpreter (i.e.; libpython3.9.so.1.0 on Linux or python37.dll on Windows)
- Bundles all this with a stub that first unpacks these files to disk or memory and then executes the original python code with the included interpreter.

While there are a few projects that can create such packaged executables, the most well-known is PyInstaller.

To understand how packaging works, we create a packed python executable on Windows. Figure 1 shows an extremely sophisticated example program that requires a master's degree in computer engineering and around 10 years of experience to create.

Figure 1 - Test program "evil_program.py"

We can easily run this program in a Windows terminal, as shown in Figure 2.

Figure 2 - Running evil_program.py

To turn this python program into a packaged EXE file we can use PyInstaller, which I installed in a python virtual environment (Figure 3).

Figure 3 - Creating an EXE with PyInstaller

It is worth your time to scroll through the logs since they give some insight into what PyInstaller does under the hood. Once finished, the newly created evil_program.exe is listed under the \dist\ folder. Figure 4 shows that we can run this executable and get the same result as directly running the code. The big difference is that we can now move this EXE file to another Windows machine and it should run standalone without any python dependency.

Figure 4 - Running the newly created EXE file

## Unpacking python < 3.9 on Windows

Now that we have a packed EXE file, we can try to revert it back to python source code. In a real reverse engineering scenario, the first question is usually, “how do we find out that the analyzed binary is a packed python program?” The most common clue is that we will see a lot of strings starting with py (Figure 5).

Figure 5 - Searching for 'py' in the binary's strings

Specifically to PyInstaller, we will also see the string MEIPASS in the binary.

The second question is, “which python version is used by the program?” The easiest way to find this out is to run the program and monitor what files are created in the operating system’s (OS) temporary folder. That’s because PyInstaller first unpacks all files in the temporary folder. By monitoring the filesystem activity, we can see that the python38.dll (Figure 6) is saved in the temporary folder. This tells us that python 3.8 was used to create the packed program and therefore we will need the same python version for all further analysis.

Figure 6 - Monitoring filesystem activity

To recover the source code, we have to tackle two challenges:

1. Unpack all files from the EXE file. That will give us compiled python bytecode (.pyc) files
2. Decompile the interesting .pyc files

The process of unpacking the EXE file will be similar in all versions of python under all operating systems. The bigger challenge is decompiling the .pyc files, because that changes in every python version and tools only work with specific versions.

For unpacking this EXE file, we will use pyinstxtractor. Just download the pyinstxtractor.py to the folder where you want to work with it (Figure 7 shows how to do that). An important detail to note is that the python interpreter used must be the same version as the packed python program.

Figure 7 - Unpacking evil_program.exe

Another important detail is that pyinstxtractor also provides hints as to which files could be the main file of the python program. There are often some false positives, but this is still a huge help if the analyzed project is big. In this case, we know that the main file is evil_program.pyc.

The EXE is unpacked into the evil_program.exe_extracted folder (Figure 8).

Figure 8 - Extracted .pyc files

The next step is to decompile the evil_program.pyc. For that, we will use a tool called uncompyle6. Again, this is a point where one must be conscious about the python version and consult the documentation of the tool being used for decompilation. Uncompyle6 only supports up to python 3.8. After that, you will have to look for another tool (which we will discuss in the next section). The decompilation process is shown in Figure 9.

Figure 9 - Decompiling the evil_program.pyc

With that, we have reached our goal and recovered the source code of this simple packed python program.

## Unpacking python >= 3.9 on Linux

In this section, we are going to go through the same process under Linux using a newer python version. The file we will analyze is a real malware sample that we found on VirusTotal during our recent threat hunting. More information about this binary can be found on VirusTotal. Once I reverse engineered the file, I thought it would be interesting to write a blog post about the unpacking process. We are not going to focus on the analysis of the sample in this post.

To unpack the sample, we again use pyinstxtractor, but with a twist. Figure 10 shows that the sample is a 64-bit ELF binary. We cannot use pyinstxtractor directly on the ELF binary. So, we first need to dump the pydata section of the file into a separate file and run pyinstxtractor on that.

Figure 10 - Dump of the pydata section

The unpacking is shown in Figure 11. Again, we need to be conscious of using the correct python version, which in this case is 3.9.

Figure 11 - Unpacking the pydata.dump

The fact that there is a RansomWare.pyc in the unpacked data makes it obvious what we are dealing with.

With python 3.9 we can no longer use uncompyle6. Instead, we can use a tool like Decompyle++, which is a very promising project that uses a different, more generic, approach to decompilation. However, building the project is not very well explained on the website, so Figure 12 shows you how to download and build it.

Figure 12 - Building Decompile++ pycdc

To call the pycdc command from anywhere, we can also run sudo make install.

The pycdc command is the decompiler, so we use that to recover the source code of the RansomWare.pyc, as shown in Figure 13.

Figure 13 - Decompiling the RansomWare.pyc

With that, we have reached our goal of recovering most of the original source code of RansomWare.py. Unfortunately, we may also see functions like the one in Figure 14, where decompilation failed at some point.

Figure 14 - Failed to decompile the write_key() function

This also happens in Java and .Net when we decompile the bytecode. Sometimes, the decompiler fails and we only get partial code. In such cases, we need to find other ways to determine what happened in that function, such as dynamic analysis. In this case, we can use the pycdas command to recover the 'disassembled' bytecode. There we can look up the functions, where the decompilation failed. Figure 15 shows the bytecode disassembly of the write_key() function.

Figure 15 - Bytecode disassembly of the write_key() function

At the beginning of this section, I mentioned that we won't be analyzing the sample. It looks like ransomware, the python code was written for Windows but packed as an ELF executable, which usually runs on Linux/Unix systems. This may indicate that the sample is intended for the Windows Subsystem for Linux (WSL). But that's a story for another blog post.

## Conclusion

In this blog post we covered how to unpack and decompile python programs packaged with PyInstaller. We also discussed the following scenarios:

- Windows
- Linux
- Python versions greater or equal to 3.9
- Python versions lower or equal to 3.8

Reverse engineering python malware can be very useful because we can analyze it at a source code level, which of course is much more efficient.

## Fortinet Protection

The analyzed ransomware sample in this blog is detected by the following (AV) signature:

ELF/Filecoder.IG!tr

Since PyInstaller writes the unpacked files on the disk before executing it, FortiEDR is also able to identify the malicious content.

## IOCs

For the discussed ransomware sample:

*Hxxps://images[.]idgesg[.]net/images/article/2018/02/ransomware_hacking_thinkstock_903183876-100749983-large[.]jpg*
*lynrx_at_protonmail[.]com*
*fernet_key.txt*
*EMAIL_ME.txt*

*Learn more about Fortinet's FortiGuard Labs threat research and intelligence organization and the FortiGuard Security Subscriptions and Services portfolio.*

## Related Posts