Exploiting insecure file extraction in Python for code execution

Intro to Computer Security



Mohammed Faisal Khan

# Summary:

Compressed file extraction with insecure code vulnerable to path traversal in Python can result in arbitrary code execution by overwriting \_\_init\_\_.py

# Introduction:

File uploads carry a significant risk that many are not aware of, for example, one of the easiest way to achieve code execution in PHP is by exploiting an insecurely written file upload logic. If the code doesn’t restrict what file to be uploaded to a PHP server then by fooling the upload logic, one can upload an arbitrary PHP (.php) file to the server that can be used to execute an arbitrary PHP code.

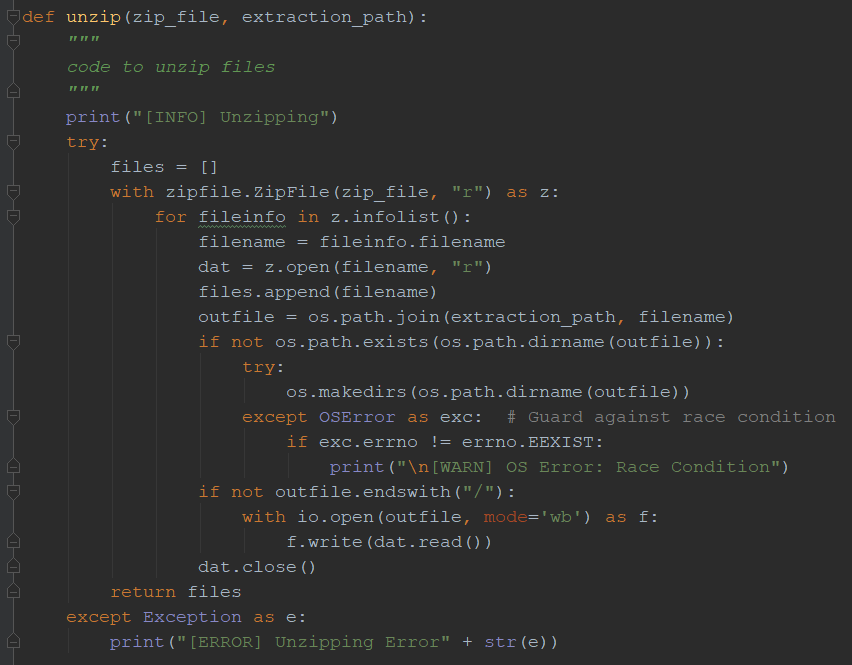
Modern web frameworks written in Go, Node.js, Python, Ruby etc. don’t work in the same way i.e. even if we manage to upload a Python (.py) or JavaScript (.js) file to the server, we can cannot request anything through the URL as it is not exposed, even if we were to have access to the resource URL, it won’t trigger any code execution as the file is treated as a static file and returns plain text. But when an insecurely written upload function is fooled into uploading a .py file on a modern framework it can allow for code execution in the back end of the platform.

This report will focus on explaining one such method of overcoming an insecurely written function using python, and for this demonstration we will be using flask web framework.

# The Insecure Code:

The thumb rule for application security is to never trust user input. Meticulously crafted payloads that may seem harmless can wreak havoc on a system once it bypasses the upload code. A compressed file carefully put together that looks legit upon extraction can do harm if it is handled by insecure code. To get a grip on the understanding of what can go wrong while using an insecure code for extraction purpose we will study the code snipped shown below.

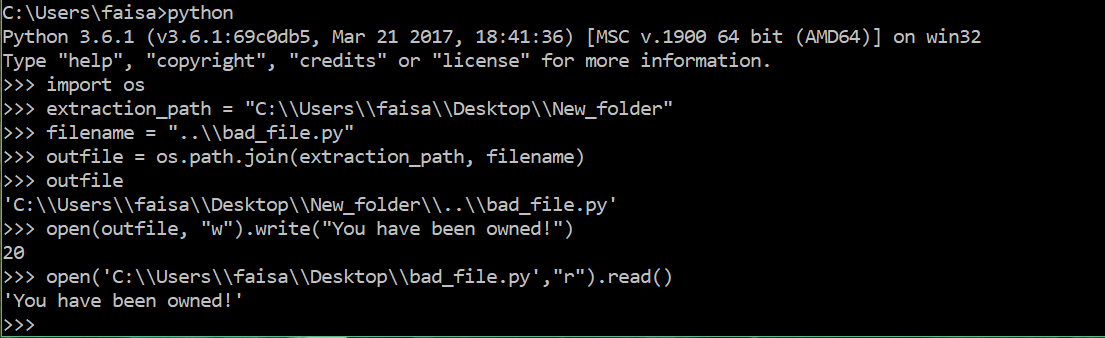
The code shown below is fairly simple and purposefully created to simulate the extract function, once the file is uploaded to the server it is send to the unzip() function.



Notice the outfile parameter here:



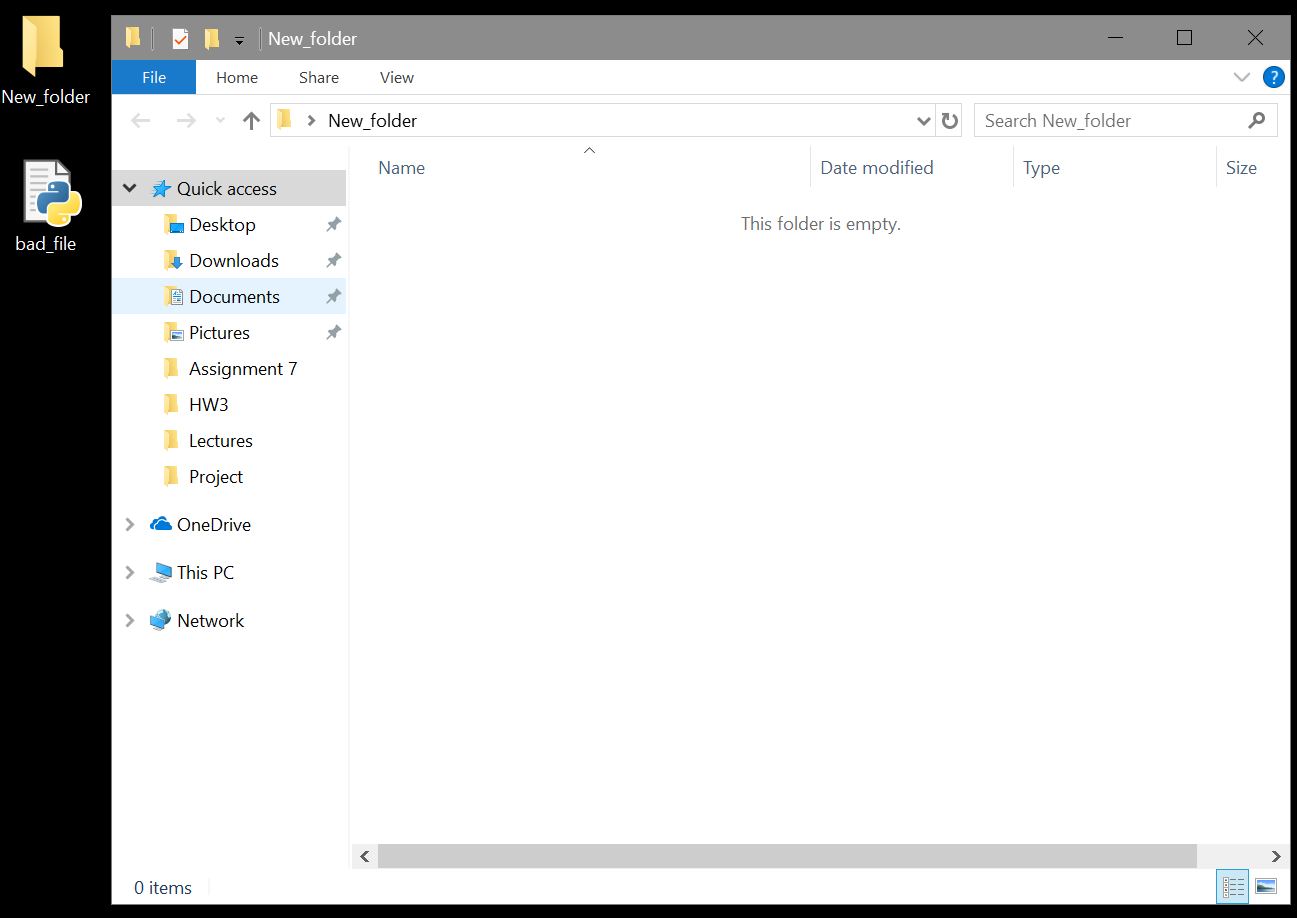
It is controlled by the user, if we were to carefully name the file such that the filename resembles a path we can extract the file to another path or put it another directory. Let’s take an example below and run this scenario in a shell.



We have specified an extraction path here as "C:\\Users\\faisa\\Desktop\\New\_folder" and we want to extract a file to this location, in our case the filename is: ..\\bad\_file.py, which closely resembles a path.

The outfile creates a new path for the file to be extracted, as seen above due to the abuse of path traversal we have a different path for the extracted file.

To put things in perspective the code is supposed to extract the file in the same directory, in this case New\_folder, but due to manipulation of path traversal we have the file in another location, in this scenario on the windows desktop as shown below.



This type of vulnerability in an extract function can have serious implications if a malicious file were to be extracted on one of the configuration file used by the system.

# Arbitrary Code Execution:

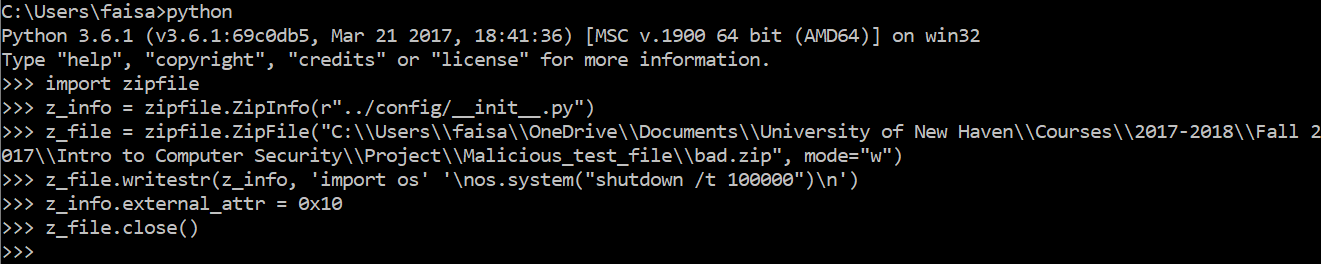
Now that we understand insecure extraction we will try to execute the malicious code, we have written a program using Python Flask. We will use \_\_init\_\_.py in Python to achieve code execution. From the document.

The \_\_init\_\_.py files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, \_\_init\_\_.py can just be an empty file, but it can also execute initialization code for the package or set the \_\_all\_\_ variable, described later.

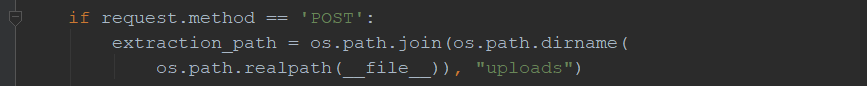
So, if we can overwrite \_\_init\_\_.py file with arbitrary Python code inside a directory of the web application that acts as a package, then we can achieve code execution if that package is imported by the application. For our code to execute, a server restart is required in most case. But in this example, we are running a Flask server with debug set to True which means every time a Python file is changed, the server will do a restart.

# Crafting the Payload:

The vulnerable web app has a directory called config. There is already a \_\_init\_\_.py and a settings.py in this directory. The main server file server.py imports settings.py from config directory, which means if we can write code into config/\_\_init\_\_.py, we will be able to achieve code execution. We can craft the payload using the following code:



If you consider the file upload code, you can see that the file uploads are extracted into uploads directory and this where we normally want to extract the file.



We can create a malicious filename with zipfile.ZipInfo(). Here we give the filename as ../config/\_\_init\_\_.py to overwrite \_\_init\_\_.py inside config directory. z\_info.external\_attr = 0x10 (Use z\_info.external\_attr = 0777 << 16L for Linux environment) will set the file permission to read and write by everyone. Let's create a zip file and upload it to the vulnerable web app.

A screenshot of a computer

Description generated with very high confidence

Once the payload is uploaded to the server and the zip file is extracted, we can see that the Flask app reloads and the server console prints test. This indicates that the code execution is successful.

# Exploiting Real World Applications:

In this scenario, the arbitrary code executed instantly as the Flask server was running on debug mode. This may not be the case elsewhere. You might need to wait until the server is restarted. Another problem is that we don't always know the package directory like config in this case. It's easy with an open source project where you have access to the source code. For closed source applications, you can try to take a good guess for package directories like conf, config, settings, utils, urls, view, tests, scripts, controllers, modules, models, admin, login etc. These are some of the common package directories found in some Python web frameworks like Django, Flask, Pyramid, Tornado, CherryPy, web2py etc.

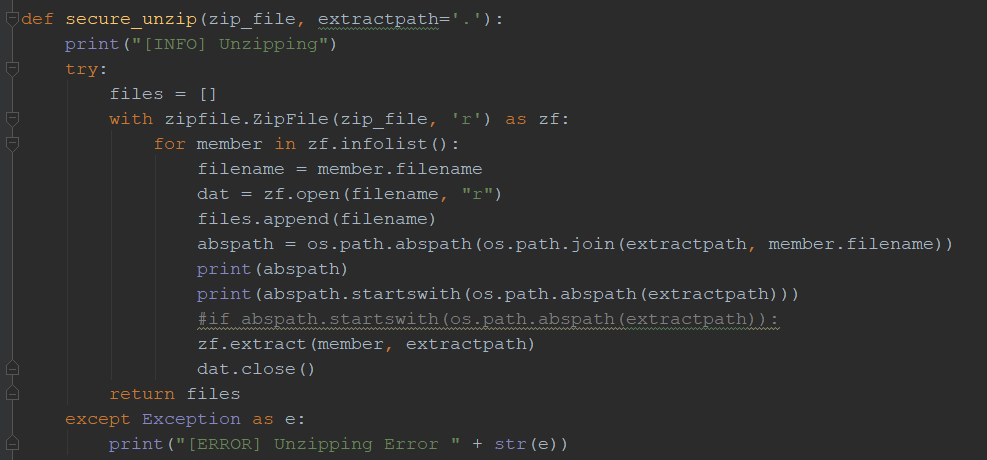
Alternatively, let's say the web application is running inside Ubuntu Linux. The installed and inbuilt Python packages will be available under: /home/<user>/.local/lib/python2.7/site-packages/pip. Assuming that the app is running under user directory, you can craft a filename like ../../.local/lib/python2.7/site-packages/pip/\_\_init\_\_.py. Upon extraction, this creates \_\_init\_\_.py file inside pip directory. If the app is using virtualenv and let's say the virtualenv directory is venv, you can use a filename like ../venv/lib/python2.7/site-packages/pip/\_\_init\_\_.py. This will brick pip, but next time someone runs pip command in the server, your code will execute!

# Secure Code to Prevent Exploit:

To prevent this vulnerability, you should use ZipFile.extract() for extracting files. The zipfile documentation says:

If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: ///foo/bar becomes foo/bar on Unix, and C:\foo\bar becomes foo\bar on Windows. And all ".." components in a member filename will be removed, e.g.: ../../foo../../ba..r becomes foo../ba..r. On Windows illegal characters (:, <, >, |, ", ?, and \*) replaced by underscore (\_).

This will prevent any kind of exploit using path traversals.



# References:

1. <https://github.com/MobSF/Mobile-Security-Framework-MobSF/issues/358>
2. <https://github.com/ajinabraham/bad_python_extract/blob/master/server.py#L62-L63>
3. <https://docs.python.org/2/library/zipfile.html#zipfile.ZipFile.extract>
4. <https://stackoverflow.com/questions/434641/how-do-i-set-permissions-attributes-on-a-file-in-a-zip-file-using-pythons-zip>
5. <https://ajinabraham.com/blog/exploiting-insecure-file-extraction-in-python-for-code-execution>
6. https://www.lifewire.com/shutdown-command-2618100