Catch Me If You Can: Bypassing Malicious Package Detectors Through API Obfuscation

Biagio Montaruli
biagio.montarul@gmail.com

$ whoami

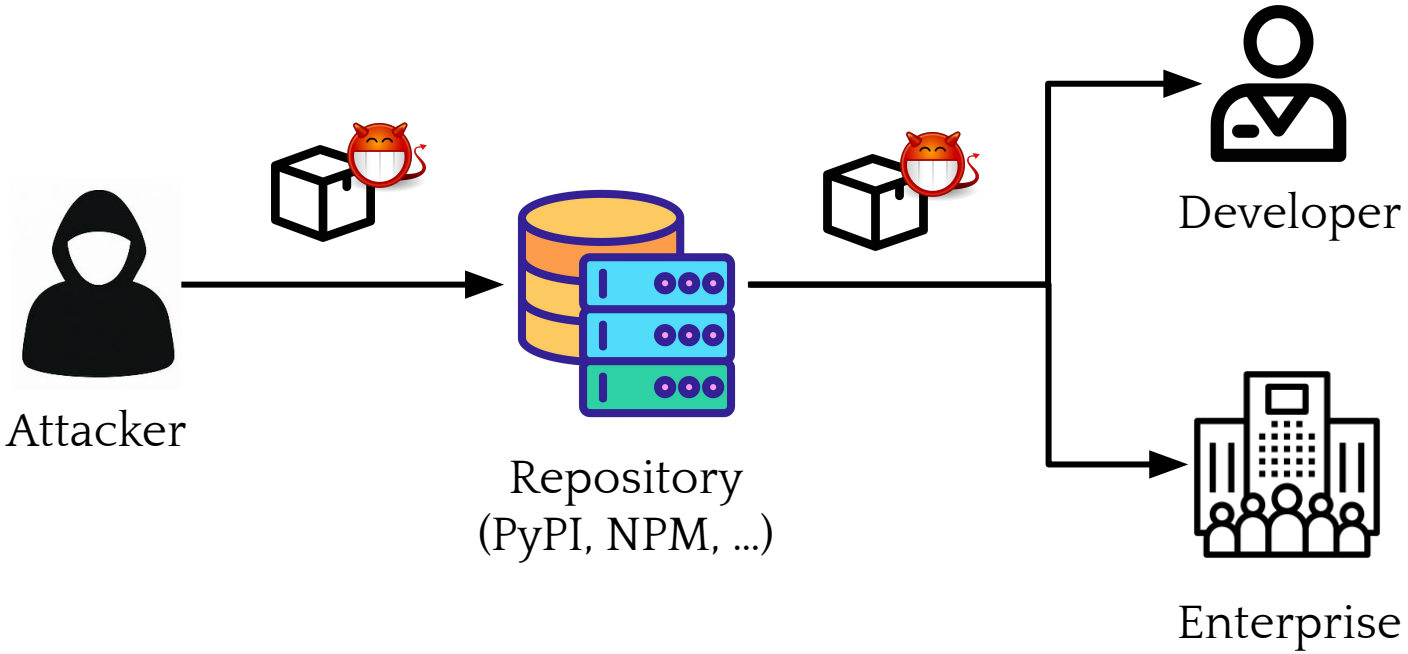
- AI security researcher @ SAP Labs France & EURECOM
  - PhD almost done 🤞

- My research topics:
  - Software supply chain security
  - macOS malware detection
  - Phishing
  - Web Application Firewalls
  - ... mix all together with AI

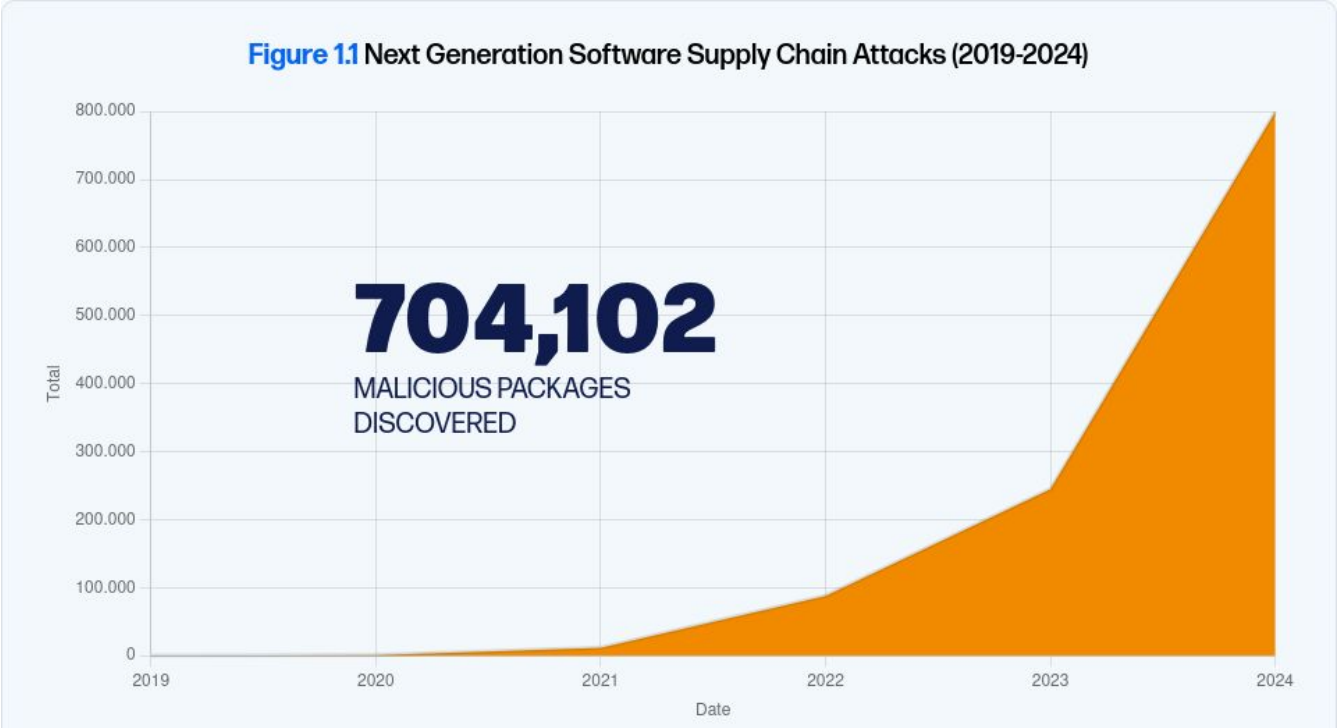
biagio-montaruli

biagiom

Software supply chain attacks



The rise of software supply chain attacks



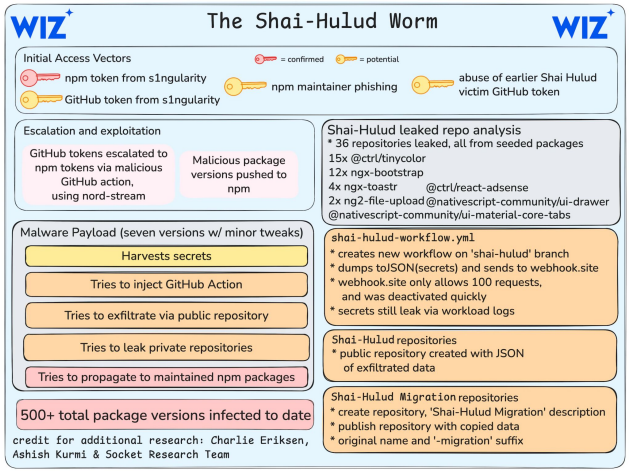
Source: 2024 State of the Software Supply Chain Report, Sonatype

The software supply chain is the New Attack Surface

Widespread Supply Chain Compromise Impacting npm Ecosystem

Release Date: September 23, 2025

CISA is releasing this Alert to provide guidance in response to a widespread software supply chain compromise involving the world's largest JavaScript registry, npmjs.com. A self-replicating worm—publicly known as "Shai-Hulud"—has compromised over 500 packages.[i]

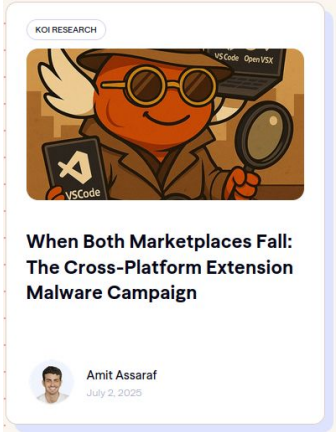


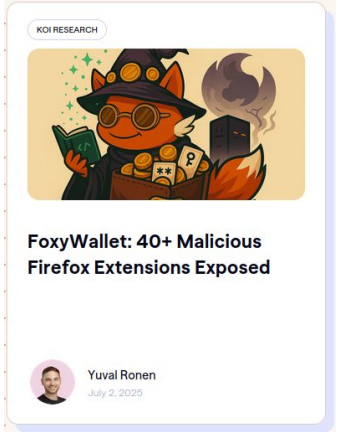
Python Status
@PythonStatus

Monitoring: New user registration on PyPI is temporarily suspended. The volume of malicious users and malicious projects being created on the index in the past week has outpaced our ability to respond to it in a timely fashion, especi...

status.python.org
PyPI new user registration temporarily suspended
New user registration on PyPI is temporarily suspended. The volume of malicious users and malicious projects bei...

9:11 PM · Dec 27, 2023 · **265** Views





Source: Koi

Detection of supply chain attacks

Static analysis

- Fast, lightweight, scale very well
- Look for IoC, suspicious APIs, ...
- Many tools and ML solutions based on static features: GuardDog, OSSGadget, ...

Dynamic analysis

- Run the package in a VM/sandbox
- Trace execution: network, filesystem operations
- Example: OSSF Package Analysis

Semgrep

The Problem: Pattern matching can't stop what it can't see

Detectors heavily rely on static detection:

- Fast, lightweight, scale very well
- Simple pattern-matching

API pattern matching can be easily bypassed by exploiting the Python's polymorphism

What if the attacker uses:

```
getattr(__import__('subrprocess'), 'call')(...)
```

```
rules:
  - id: code-execution
    languages:
      - python
    message: This package is executing OS commands in the setup.py file
    metadata:
      description: Identify when an OS command is executed in the setup.py file
    patterns:
      # exec argument must be hardcoded string
      - pattern-either:
          - patterns:
              - pattern: exec("...", ...)
              - pattern: exec($ARG1, ...)
          - patterns:
              - pattern: exec("...". ...)
              - pattern: exec($ARG1. ..., ...)
          - patterns:
              - pattern: exec("..." + ...)
              - pattern: exec($ARG1 + ..., ...)

          # subprocess module
          - pattern: subprocess.getoutput($ARG1, ...)
          - pattern: getoutput($ARG1, ...)
          - pattern: subprocess.getoutput([..., "... $ARG1 ...", ...], ...)
          - pattern: getoutput([..., "... $ARG1 ...", ...], ...)

          - pattern: subprocess.call($ARG1, ...)
          - pattern: call($ARG1, ...)
          - pattern: subprocess.call([..., "... $ARG1 ...", ...], ...)
          - pattern: call([..., "... $ARG1 ...", ...], ...)
```

API Obfuscation: 3 key pillars

Importing a module

- Default:

```
import module
```

- Obfuscated (inline import):

```
__import__('module')
```

Function call

- Default:

```
function(...)
```

- Obfuscated (using `__call__`):

```
function.__call__(...)
```

Referencing a module's function

- Default:

```
import module
...
module.function(...)
```

Obfuscated variants:

- Using `__dict__` special method:

```
module.__dict__['function']
```

- Using `__getattribute__` special method:

```
module.__getattribute__('function')
```

- Using `getattr()` function:

```
getattr(module, 'function')
```

API Obfuscation: Connecting the dots

```
os.system(<PAYLOAD>)
```

```
os.system.__call__(<PAYLOAD>)
os.__dict__['system'](<PAYLOAD>)
os.__dict__['system'].__call__(<PAYLOAD>)
os.__getattribute__('system')(<PAYLOAD>)
os.__getattribute__('system').__call__(<PAYLOAD>)
__import__('os').system(<PAYLOAD>)
__import__('os').system.__call__(<PAYLOAD>)
__import__('os').__dict__['system'](<PAYLOAD>)
__import__('os').__dict__['system'].__call__(<PAYLOAD>)
__import__('os').__getattribute__('system')(<PAYLOAD>)
__import__('os').__getattribute__('system').__call__(<PAYLOAD>)
getattr(os, 'system')(<PAYLOAD>)
getattr(os, 'system').__call__(<PAYLOAD>)
getattr(__import__('os'), 'system')(<PAYLOAD>)
getattr(__import__('os'), 'system').__call__(<PAYLOAD>)
```

Catching API Obfuscation: a new rule for GuardDog

guarddog / guarddog / analyzer / sourcecode / **api-obfuscation.yml**

biagiom API obfuscation: new rule and test case ✓ — f2f3cd5 · 2 months ago — History

Code | Blame — 42 lines (41 loc) · 2.09 KB — Raw

```yaml
rules:
    - id: api-obfuscation
      languages:
        - python
      message: This package uses obfuscated API calls that may evade static analysis detection
      metadata:
        description: Identify obfuscated API calls using alternative Python syntax patterns
      severity: WARNING
      patterns:
        - pattern-either:
          # Covered cases:
          # 1) __dict__ access patterns: $MODULE.__dict__[$METHOD](...) / .__call__(...)
          # 2) __getattribute__ patterns: $MODULE.__getattribute__($METHOD)(...) / .__call__(...)
          # 3) getattr patterns: getattr($MODULE, $METHOD)(...) / .__call__(...)
          # It also covers the case where $MODULE is imported as __import__('mod')
          - patterns:
              - pattern-either:
                  - pattern: $MODULE.__dict__[$METHOD]($...ARGS)
                  - pattern: $MODULE.__dict__[$METHOD].__call__($...ARGS)
                  - pattern: $MODULE.__getattribute__($METHOD)($...ARGS)
                  - pattern: $MODULE.__getattribute__($METHOD).__call__($...ARGS)
                  - pattern: getattr($MODULE, $METHOD)($...ARGS)
                  - pattern: getattr($MODULE, $METHOD).__call__($...ARGS)
              - metavariable-regex:
                  metavariable: $MODULE
                  regex: "^[A-Za-z_][A-Za-z0-9_\\.]*$|^__import__\\([\"']?[A-Za-z_][A-Za-z0-9_]*[\"']?\\)$"
              - metavariable-regex:
                  metavariable: $METHOD
                  regex: "^[\"']?[A-Za-z_][A-Za-z0-9_]*[\"']?$"

          # --- Additional Cases: __import__('mod').method(...) / .__call__(...)
          - patterns:
              - pattern-either:
                  - pattern: __import__($MODULE).$METHOD($...ARGS)
                  - pattern: __import__($MODULE).$METHOD.__call__($...ARGS)
              - metavariable-regex:
                  metavariable: $MODULE
                  regex: "^[\"']?[A-Za-z_][A-Za-z0-9_]*[\"']?$"
              - metavariable-regex:
                  metavariable: $METHOD
                  # avoid matching __getattribute__
                  regex: "[^(__getattribute__)][A-Za-z_][A-Za-z0-9_]*"
```

Demo Time

Download my GitHub repo with PoC:

```
$ git clone https://github.com/biagiom/api_obfuscation.git && cd api_obfuscation
```

Create a virtual environment (venv) and install GuardDog v2.6.0 (without my new rule):

```
$ python3 -m venv gd-base && source gd-base/bin/activate
(gd-base) $ python3 -m pip install guarddog==2.6.0
```

Scan the original and obfuscated malicious packages:

```
(gd-base) $ guarddog pypi scan ./packages_pypi/1337c-4.4.7 --output-format=json |
python -m json.tool
(gd-base) $ guarddog pypi scan ./packages_pypi/1337c-4.4.7_obfuscated
--output-format=json | python -m json.tool
```

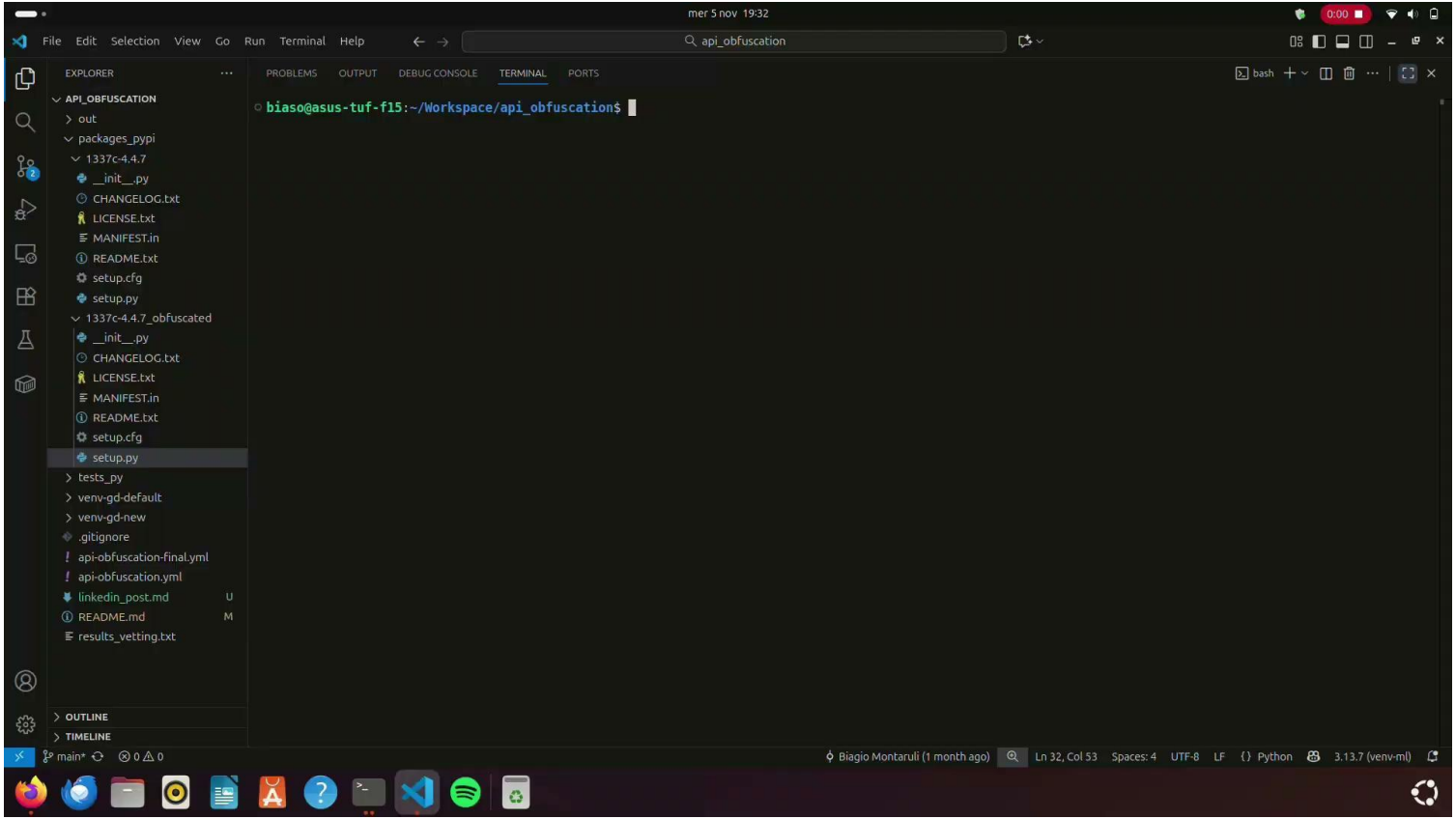
Create a new venv and install GuardDog v2.7.0 (with my new rule):

```
$ python3 -m venv gd-new && source gd-new/bin/activate
(gd-new) $ python3 -m pip install guarddog==2.7.0
```

Scan again the obfuscated package: API obfuscation is detected, right?

```
(gd-new) $ guarddog pypi scan ./packages_pypi/1337c-4.4.7_obfuscated
--output-format=json | python -m json.tool
```

Demo Time


mer 5 nov 19:32
File Edit Selection View Go Run Terminal Help
api_obfuscation
EXPLORER
API_OBFUSCATION
out
packages_pypi
1337c-4.4.7
__init__.py
CHANGELOG.txt
LICENSE.txt
MANIFEST.in
README.txt
setup.cfg
setup.py
1337c-4.4.7_obfuscated
__init__.py
CHANGELOG.txt
LICENSE.txt
MANIFEST.in
README.txt
setup.cfg
setup.py
tests_py
venv-gd-default
venv-gd-new
.gitignore
api-obfuscation-final.yml
api-obfuscation.yml
linkedin_post.md
README.md
results_vetting.txt
OUTLINE
TIMELINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash
biaso@asus-tuf-f15:~/Workspace/api_obfuscation$
main*
Biagio Montaruli (1 month ago)
Ln 32, Col 53
Spaces: 4
UTF-8
LF
Python
3.13.7 (venv-ml)

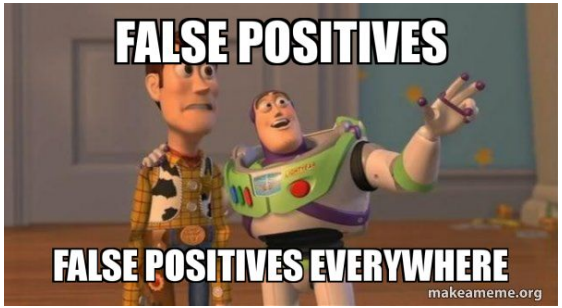
Large-scale assessment

1. How many malicious packages detected in the wild?

- Evaluated on a real-world dataset of 354 malware
  - collected from a 80-days vetting campaign
- **10 new malware previously undetected!!!**

2. How many FPs does the new rule trigger?

- Evaluated on MalwareBench
  - goodware: 2802, malware: 1981
- **FPR: 1.46%** (41 out 2802)



Key Takeaways

1) Pattern matching detection, while fast and effective for known threats, is limited against **adaptive adversaries**.

2) **Python**'s flexibility creates inherent security **blind spots**.

3) Static analysis must evolve beyond simple signature-based approaches: need for **robust signatures** and combine them with **dynamic analysis** and **ML techniques**.

4) **Detection trade-offs**: catching sophisticated attacks vs. avoiding false positives in legitimate packages.

That's all folks!

Thank you very much for attending my talk!

Demo PoC: https://github.com/biagiom/api_obfuscation.git

To contact me: biagio.montarul@gmail.com