Alpha-Omega

python
SOFTWARE FOUNDATION

UNMASKING PHANTOM DEPENDENCIES WITH SOFTWARE BILL-OF-MATERIALS AS ECOSYSTEM-NEUTRAL METADATA

AUTHOR:
SETH LARSON
PYTHON SOFTWARE FOUNDATION

SUMMARY

This work was completed by Security Developer-in-Residence Seth Larson, whose role is sponsored by Alpha Omega. Thank you to Alpha Omega for sponsoring the Python Software Foundation.

In September 2023, Endor Labs published a report on open source dependencies and coined the term "Phantom Dependency" to mean a software dependency that wasn't tracked using packaging metadata, manifests, or lock files. These dependencies, therefore, are not discoverable by common software composition analysis (SCA) tools and workflows such as vulnerability, inventory, policy, and compliance management tools.

Seth authored PEP 770, a new standard allowing Python packages to provide installers and users with metadata about software included in packages using Software Bill-of-Materials (SBOMs). The ecosystem-neutrality of SBOM documents makes them a great medium for describing both Python and non-Python software included in Python packages.

The availability of PEP 770 SBOM metadata included in package archives is expected to be rapid once the standard is adopted by build tools. PEP 770 is backwards compatible and can be enabled by default by tools, meaning most projects won't need to manually opt in to begin generating valid PEP 770 SBOM metadata.

Python is not the only software package ecosystem affected by the "Phantom Dependency" problem. The approach using SBOMs for metadata can be remixed and adopted by other packaging ecosystems looking to record ecosystem-agnostic software metadata.

WHAT ARE “PHANTOM DEPENDENCIES”?

Phantom Dependencies are software components that are included in software packages, container images, and deployments that are not referenced in any metadata. This makes those software components either difficult or impossible to discover by tools doing software composition analysis.

Within Endor Labs’ report, Python is named as one of the most affected packaging ecosystems by the “Phantom Dependency” problem. There are multiple reasons that Python is particularly affected:

- There are many methods for interfacing Python with non-Python software, such as through the C-API or FFI. Python can “wrap” and expose an easy-to-use Python API for software written in other languages like C, C++, Rust, Fortran, Web Assembly, and more.

- Python is the premier language for scientific computing and artificial intelligence, meaning many high-performance libraries written in system languages need to be accessed from Python code.

- Finally, Python packages have a distribution type called a “wheel”, which is essentially a zip file that is “installed” by being unzipped into a directory, meaning there is no compilation step allowed during installation. This is great for being able to inspect a package before installation, but it means that all compiled languages need to be pre-compiled into binaries before installation.

These three reasons combine to mean that the Python package ecosystem is particularly affected by the “Phantom Dependency” problem.

WHAT RISKS DO PHANTOM DEPENDENCIES POSE?

Using the popular package for processing images called “Pillow” as an example, we can see many libraries bundled within the wheel archives when we install the package into a virtual environment:

```
$ python -m pip install pillow
$ ls venv/lib/python3.13/site-
packages/pillow.libs

libavif-01e67780.so.16.3.0
libbrotlicommon-c55a5f7a.so.1.1.0
libbrotlidec-2ced2f3a.so.1.1.0
libfreetype-083ff72c.so.6.20.2
libharfbuzz-fe5b8f8d.so.0.61121.0
libjpeg-8a13c6e0.so.62.4.0
liblcms2-cc10e42f.so.2.0.17
liblzma-64b7ab39.so.5.8.1
libopenjp2-56811f71.so.2.5.3
libpng16-d00bd151.so.16.49.0
libsharpyuv-60a7c00b.so.0.1.1
libtiff-13a02c81.so.6.1.0
libwebp-5f0275c0.so.7.1.10
libwebpdemux-efaed568.so.2.0.16
libwebpmux-6f2b1ad9.so.3.1.1
libXau-154567c4.so.6.0.0
libxcb-64009ff3.so.1.1.0
```

Many of these libraries are well-known projects like xz-utils (liblzma), libpng, libjpeg, and libbrotli.

However, if we then scan this Python virtual environment with common software scanning tools like OSV-Scanner and Syft, they do not detect the bundled system libraries, instead only detecting the Python dependencies:

```
$ syft dir:venv/ --select-catalogers "+sbom-cataloger"

pillow          11.3.0      python
pip             24.2        python
```

None of the libraries we saw before are being discovered by these software scanning tools. What does this mean for users of open source software?

Many common software scanning tools are missing the complete picture of applications and their dependencies in terms of vulnerability scanning, policy, and compliance. For example, the Pillow Python package bundles the "libwebp" dependency, which was vulnerable to an actively exploited vulnerability (CVE-2023-4863) and was exploitable through Pillow APIs. Without the knowledge that Pillow bundles this dependency, users of the Pillow package may not know to upgrade to later versions of Pillow that update the bundled libwebp dependency to a fixed version.

HOW PERVASIVE ARE PHANTOM DEPENDENCIES?

One of the most common ways that Phantom Dependencies are included in Python packages is through "wheel repairing" tools such as auditwheel, delvewheel, and delocate. These tools bundle libraries into Python packages so they're installable on many different platforms using the wheel archive format.

To quantify how often Phantom Dependencies are included in critical projects, the top 5,000 packages on PyPI ordered by downloads were queried for whether the packages included dependencies using auditwheel. Auditwheel-included libraries are the easiest Phantom Dependencies to detect due to using a specific directory name (".libs"); therefore, these numbers represent a lower bound for the number of projects with Phantom Dependencies:

System Library	Dependent Projects	Monthly Downloads
libgcc_s	112	2,750,000,000
libstdc++	64	1,500,000,000
libz	47	1,000,000,000
libgfortran	15	650,000,000
libopenblas	6	600,000,000
libcrypto	37	580,000,000
libssl	35	580,000,000
libquadmath	13	530,000,000
libffi	5	370,000,000

Overall, auditwheel was used to include system libraries in 212 of the top 5,000 packages on PyPI.

Non-Python programming language package metadata files are another sign of Phantom Dependencies. C, C++, JavaScript, and Rust are popular programming languages to include in Python packages. Looking again at the top 5,000 projects, we see that those languages and dependencies are in frequent use:

Programming Language	Dependent Projects	Monthly Downloads
C and C++	567	10,000,000,000
JavaScript	309	4,000,000,000
Rust	95	1,700,000,000

Another source of Phantom Dependencies is through the bundling of Python software. Measuring the number of Python packages that are bundled into other Python packages is more difficult, as we don't have an explicit marker like we do for system libraries bundled using auditwheel, etc. However, there are a few commonly used tools for bundling Python libraries, such as "vendoring," which produces a tell-tale "vendor.txt" file to manage bundled dependencies.

Querying for top packages that use "vendoring," we see the tool is popular amongst packaging tools and core libraries, which are frequently installed, such as pip, pdm, pip-tools, setuptools, and poetry-core. Despite only being used by 11 of the top 5,000 projects, those 11 projects are installed collectively over 50 billion times per month.

HOW CAN SOFTWARE BILL-OF-MATERIALS FIX THE PROBLEM?

Software Bill-of-Materials (SBOMs) are documents that contain information about software, typically the names and identifiers of software components along with other metadata like version, download location, licenses, checksums, and the relationships between components. These documents can serve as a substitute for "native" package metadata about a particular software component, providing much of the same information but in an ecosystem-neutral format.

Many SCA tools already implement handling for SBOM documents, so the adoption of PEP 770 SBOM metadata should be straightforward for most tools. Some tools today already implement such functionality, such as Syft and Grype, which can use SBOM documents included in Python packages to augment vulnerability scanning.

Let's try our example with Pillow again, but this time with some included PEP 770 SBOM metadata within the package archive. To include PEP 770 metadata, we rebuild Pillow using a development version of auditwheel. The metadata is automatically generated using Let's look at the software identifiers (Package URLs or "PURLs") included in the SBOM metadata:

```
```

```
$ cat venv/.../sboms/auditwheel.cdx.json | jq | grep "purl"

"purl": "pkg:pypi/pillow@11.2.1"
"purl": "pkg:pypi/pillow@11.2.1"
"purl": "pkg:deb/ubuntu/libwebp7@1.2.2-2ubuntu0.24.04.2"
"purl": "pkg:deb/ubuntu/libwebpdemux2@1.2.2-2ubuntu0.24.04.2"
"purl": "pkg:deb/ubuntu/libbsd0@0.11.5-1"
"purl": "pkg:deb/ubuntu/libtiff5@4.3.0-6ubuntu0.10"
"purl": "pkg:deb/ubuntu/libxcb1@1.14-3ubuntu3"
"purl": "pkg:deb/ubuntu/libxdmcp6@1%3A1.1.3-0ubuntu5"
"purl": "pkg:deb/ubuntu/libjbig0@2.1-3.1ubuntu0.24.04.1"
"purl": "pkg:deb/ubuntu/libpng16-16@1.6.37-3build5"
"purl": "pkg:deb/ubuntu/libmd0@1.0.4-1build1"
"purl": "pkg:deb/ubuntu/libwebpmux3@1.2.2-2ubuntu0.24.04.2"
"purl": "pkg:deb/ubuntu/liblcms2-2@2.12~rc1-2build2"
"purl": "pkg:deb/ubuntu/libbrotli1@1.0.9-2build6"
"purl": "pkg:deb/ubuntu/libdeflate0@1.10-2"
"purl": "pkg:deb/ubuntu/libbrotli1@1.0.9-2build6"
"purl": "pkg:deb/ubuntu/libzstd1@1.4.8%2Bdfsg-3build1"
"purl": "pkg:deb/ubuntu/libfreetype6@2.11.1%2Bdfsg-
1ubuntu0.3"
"purl": "pkg:deb/ubuntu/libjpeg-turbo8@2.1.2-0ubuntu1"
"purl": "pkg:deb/ubuntu/libxau6@1%3A1.0.9-1build5"
"purl": "pkg:deb/ubuntu/libopenjp2-7@2.4.0-6ubuntu0.3"
```

This list of software identifiers is included in the SBOM generated by auditwheel. Now that this SBOM data exists within the virtual environment, software scanners are able to properly detect the full set of dependencies within the virtual environment (no more Phantom Dependencies!):

```
$ syft dir:venv/ --select-catalogers "+sbom-
cataloger"
NAME              VERSION                  TYPE
libbrotli1        1.0.9-2build6            deb
libbsd0           0.11.5-1                 deb
libdeflate0       1.10-2                   deb
libfreetype6      2.11.1+dfsg-1ubuntu0.3   deb
libjbig0          2.1-3.1ubuntu0.24.04.1   deb
libjpeg-turbo8    2.1.2-0ubuntu1           deb
liblcms2-2        2.12~rc1-2build2         deb
libmd0            1.0.4-1build1            deb
libopenjp2-7      2.4.0-6ubuntu0.3         deb
libpng16-16       1.6.37-3build5           deb
libtiff5          4.3.0-6ubuntu0.10        deb
libwebp7          1.2.2-2ubuntu0.24.04.2   deb
libwebpdemux2     1.2.2-2ubuntu0.24.04.2   deb
libwebpmux3       1.2.2-2ubuntu0.24.04.2   deb
libxau6           1:1.0.9-1build5          deb
libxcb1           1.14-3ubuntu3            deb
libxdmcp6         1:1.1.3-0ubuntu5         deb
libzstd1          1.4.8+dfsg-3build1       deb
pillow            11.2.1                   python
pip               25.1.1                   python
```

Thanks to the automatically generated PEP 770 SBOM metadata from auditwheel our scanner was now able to detect all the bundled software libraries by their Package URL and therefore should be able to fetch vulnerability data and alert us when our software is not secure and needs upgrading.

SUSTAINABLE IMPLEMENTATION APPROACH

When designing a new package metadata standard, one of the top concerns is reducing the amount of effort required from the mostly volunteer maintainers of packaging tools and the thousands of projects being published to the Python Package Index. The design of PEP 770 was iteratively created, starting with an existing standard that had been adopted by packaging tool maintainers for license files (PEP 639), which was similar in terms of including one or more optional metadata files inside a package archive.

However, PEP 770 had the fortune of being able to learn from PEP 639's pain points during implementation. For example, PEP 639 required a new Metadata Field and therefore required incrementing the "Metadata Version". Any time a metadata version is incremented in the Python packaging ecosystem the process can take a while for PyPI, packaging core libraries, and build backends to incrementally and carefully adopt the new metadata field and version to avoid breaking users that are using the latest versions of their tools.

With PEP 770, we wanted to avoid that issue if possible. Working with packaging PEP reviewers and querying the same dataset of Python package contents, we were able to safely create a new mechanism for including files in wheels using reserved directory names under the ".dist-info" directory. By defining PEP 770 SBOM metadata as using a directory of files, rather than a new metadata field, we were able to side-step all the implementation pain.

To ease adoption for projects, we submitted patches to the most critical projects that are capable of generating PEP 770 SBOM data about dependencies, such as auditwheel and vendoring. More projects and their users have signalled their interest in adopting PEP 770, such as CMake and Meson.

FUTURE WORK

As build tools begin adopting PEP 770, the availability of more accurate SBOM data of Python packages will be available for analysis and scanning tools. We'll be working to submit issues on popular open source SBOM and vulnerability scanning tools, and gradually, Phantom Dependencies will become less of an issue for the Python package ecosystem.

Having spoken to other open source packaging ecosystem maintainers, we have come to learn that other ecosystems have similar issues with Phantom Dependencies. We welcome other packaging ecosystems to adopt Python's approach with PEP 770 and are willing to provide guidance on the implementation.