software stewardship lab

PROPOSAL · 11 NOV 2026

Securing Python's Binary Dependencies



PRINCIPAL INVESTIGATOR
Vlad Miricioiu



ADVISER
Andrew Nesbitt



ADVISER
Dawn Foster

This is a proposal to bring impactful security improvements, such as augmented CVE reachability data, to the Python ecosystem by tracking currently-invisible binary dependencies.

A large amount of infrastructure relies on Python for its ability to easily interface with compiled libraries, particularly AI and scientific software. However, package management tooling in the Python ecosystem does not record these *binary dependency* relationships, which most frequently occur when Python code calls compiled libraries written in languages such as C or Rust.

This knowledge gap results in a *phantom dependency* — a dependency relationship that is hidden from existing data — meaning that developers cannot track or remediate CVEs in these dependencies; cannot know when these libraries become unmaintained "Bernies";[12] and cannot offer support to the maintainers of their binary dependencies.

Given that around 20.4% of popular Python packages include binary dependencies,[8] this status quo puts critical Python-based technology, from AI to built infrastructure, at risk of cyberattacks and degradation.

An effective solution to this problem must include the following elements:

New tools	Create tools to identify binary dependency relationships in existing software.
New records	Augment specifications to create a place for binary dependency relationships to be recorded, eg in package manager manifests.
Integration	Connect new dependency data to existing data sources, resulting in meaningful security improvements like reachability of previously-invisible CVEs.
Education	Create educational materials to enhance developers' cultural awareness of the problems phantom dependencies pose.

These elements must be designed such that they can be **incorporated as defaults** into existing package managers, so users do not need to opt in to receive security benefits.

This proposal outlines such a solution for the Python ecosystem that (1) can make a large positive impact with a comparatively small investment, and (2) is structured so that it can be, in principle, ported to other language ecosystems in the future.

New tools

This project will involve building a tool that can identify existing binary dependencies within Python packages, in a manner that is accurate and performant, and easy to integrate with existing tooling. Such a tool requires the following components:

1. **Dynamic library mining** — Since Python code most frequently includes binary dependencies using extension modules,[7] a Python package's binary dependencies will almost always be vendored as dynamic library (eg ELF[15]) files within the Python package's wheel releases.[14] These dynamic libraries should be mined to identify the names of required compiled libraries.
2. **Symbol mining** — Library names are insufficient to reliably link a dynamic library with the version and project it originated from. However, the symbols within these dynamic libraries have the potential to reliably identify the provenance of the compiled code, if we create a database mapping symbol patterns to project PURLs.[13] I have already begun compiling such a database, currently containing 39 million symbols found inside Python wheels.[10] ecosyste.ms[11] has confirmed they are willing to integrate and host this data.

However, these components must be developed with the following considerations in mind:

1. **Policy agnosticism** — This project should not be contigent on being accepted into any given package manager, standard or PEP, as those processes can take a long time.
2. **Simple APIs** — This new tooling should be easy to integrate into existing package management infrastructure at the infrastructure maintainers' discretion, and also usable by AI agents.
3. **Data independence** — When package registry operators do integrate this new tooling, they should not have to rely on databases or APIs hosted outside their registry's infrastructure.

The decision to build a new tool comes after a thorough review of existing solutions, which fail to meet all of the above criteria: see *auditwheel*,[5] *elfdeps*[3] and *fromager*.[6] For more detailed technical information, see my EuroPython 2026 talk[8] and my article on how binary dependencies work.[7]

New records

Once existing binary dependencies have been identified, they must be recorded, and the best location for this is in *pyproject.toml* based on the proposal in PEP 725.[2] These records would allow registries and researchers to analyse binary dependency relationships and their security status, and to enrich SBOMs and store them as specified in PEP 770.[4] I am in contact with the authors of these PEPs and will coordinate with them.

I propose creating an independent solution to enrich *pyproject.toml* with this information, that is designed to be trivial to incorporate into upstream tools such as *uv* or *pip*.

Integration

Once binary dependencies are identified, package managers can look up CVEs for those dependencies and alert users when they are vulnerable. I would prototype such a solution as part of this grant, which would provide a direct and immediate security benefit to Python users.

New education

Developers must understand the importance of such security tooling if they are to widely adopt it. As part of this project, I will create articles to educate developers on the aforementioned issues. For a more detailed explanation, see my Community Over Code keynote.[9]

Future work: deferring to system package managers

Ideally, binary dependencies would be managed not by Python infrastructure, but by the system package managers that are best-equipped to deal with them. To achieve this, the vendored binary dependencies of installed packages could be rewritten to point to versions provided by the system package manager. This would bring large security benefits by eliminating the precarity introduced by vendored binaries altogether; but would require a separate grant.

Impact projection

Because around 20.4% of popular Python packages include binary dependencies,[8] a large percentage of Python users are vulnerable to the security issues introduced by the current haphazard vendoring of binary dependencies. Currently, there is no information about CVEs in these dependencies, no information about their provenance and whether they are up-to-date, and no easy mechanism to update them.

On completion, this project would provide the tooling necessary for a solution to this problem. An integration with CVE databases would give developers full vulnerability information, and an integration with Python package managers would make these benefits the default. In the long term, this solution could be ported to other languages' package ecosystem, as it is not fundamentally bound to Python.

Importantly, the new dependency data generated by this work will make both developers and AI agents better informed regarding their dependency choices. When choosing a package, it is important to know whether this choice would introduce transitive binary dependencies, but this valuable information is currently not trivial to obtain.

Risk factors

There are two points of uncertainty to be answered during the development of this project:

- The aim is to create a solution that is as easy as possible for package manager maintainers to integrate, and the maintainers of *uv* seem open to such improvements generally; but there is no guarantee that our solution will be integrated as a default.
- A symbol database will be required to determine which version of which package shipped a dynamic library, and ecosyste.ms[11] is willing to host this data; but how can we give package registry administrators autonomy over this data? I will investigate providing the data as static Parquet files,[1] offering maintainers maximum autonomy.



Vlad Miricioiu vlad.website
Software Stewardship Lab stewardshiplab.org

Appendix A: References

[1] Apache Software Foundation. Parquet. Retrieved November 10, 2026 from https://parquet.apache.org/

[2] Pradyun Gedam, Jaime Rodríguez-Guerra, and Ralf Gommers. 2023. PEP 725 – Specifying external dependencies in pyproject.toml. Retrieved November 10, 2026 from https://peps.python.org/pep-0725/

[3] Christian Heimes. elfdeps. Retrieved November 10, 2026 from https://github.com/python-wheel-build/elfdeps/?tab=readme-ov-file

[4] Seth Larson. 2025. PEP 770 – Improving measurability of Python packages with Software Bill-of-Materials. Retrieved November 10, 2026 from https://peps.python.org/pep-0770/

[5] Robert T. McGibbon. auditwheel. Retrieved November 10, 2026 from https://github.com/pypa/auditwheel/

[6] Mark McLoughlin and Doug Hellmann. fromager. Retrieved November 10, 2026 from https://github.com/python-wheel-build/fromager

[7] Vlad Miricioiu. 2026. How Binary Dependencies Work Across Different Languages. Retrieved November 10, 2026 from https://vlad.website/how-binary-dependencies-work/

[8] Vlad Miricioiu. 2026. *Binary Dependencies: Identifying the Hidden Packages We All Depend On*. Retrieved November 10, 2026 from https://www.youtube.com/watch?v=qooQD1tLVTg

[9] Vlad Miricioiu. 2026. *Keynote: Unmasking Phantoms in our Supply Chains - Community Over Code Glasgow 2026*. Retrieved November 10, 2026 from https://www.youtube.com/watch?v=zWUgJGG8854

[10] Vlad Miricioiu. bindep. Retrieved November 10, 2026 from https://codeberg.org/stewardshiplab/bindep

[11] Andrew Nesbitt and Benjamin Nickolls. 2022. ecosyste.ms. Retrieved from https://ecosyste.ms/

[12] Andrew Nesbitt. 2026. Weekend at Bernie's. Retrieved November 10, 2026 from https://nesbitt.io/2026/05/08/weekend-at-bernies.html

[13] The purl authors. Package-URL (PURL) Specification. Retrieved November 10, 2026 from https://github.com/package-url/purl-spec

[14] PyPA. Wheel distribution format - Python Packaging User Guide. Retrieved November 10, 2026 from https://packaging.python.org/en/latest/specifications/binary-distribution-format/

[15] Wikipedia contributors. 2026. Executable and Linkable Format. Retrieved November 10, 2026 from https://en.wikipedia.org/w/index.php?title=Executable_and_Linkable_Format&oldid=1384341723

Appendix B: Previously Funded Project Summary

I have not received cash funding or AWS Promotional Credits directly or indirectly from Amazon within the past 3 years

Appendix C: LLM Disclosure

No LLMs were used for the creation of this proposal.