Alpha-Omega

python
SOFTWARE FOUNDATION

SLIPPERY ZIPS AND STICKY TAR-PITS: SECURITY AND ARCHIVES

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.

Software packages, such as Python “wheels”, Java “JARs”, and Ruby “gems”, are a set of one or more files bundled together in a standardized format. Packaging ecosystems frequently use an existing standard, such as ZIP or tar, for their archive format. These formats were created a long time ago, 36 and 46 years ago, respectively. Computing has advanced substantially since these archive formats were first introduced, but maintaining backwards compatibility means that archive features and bugs are often supported for extended periods of time. Complexities in implementing archive formats or handling user-supplied archive files can lead to an exploitable surface for attackers.

The Python language and packaging ecosystem have seen multiple ZIP and tar archive-related CVEs since 2024. This report highlights the commonalities between these vulnerabilities and details Python and PyPI's remediations of the vulnerabilities and makes recommendations for handling archive formats more securely for Python and other programming languages and packaging ecosystems.

VULNERABILITIES

This white paper covers the following vulnerabilities:

CVE ID	Project	Severity	Description
CVE-2025-8291	CPython	MEDIUM	ZIP64 End of Central Directory Locator offset not checked
CVE-2025-8194	CPython	HIGH	Infinite loop during tar extraction with negative GNU member offset.
CVE-2025-4517	CPython	CRITICAL	Tar extraction filter bypass: arbitrary writes to filesystem
CVE-2025-4330	CPython	HIGH	Tar extraction filter bypass: symlink targets outside extraction directory
CVE-2025-4138	CPython	HIGH	Tar extraction filter bypass: creating arbitrary symlinks outside extraction directory
CVE-2024-12718	CPython	MEDIUM	Tar extraction filter bypass: modifying file metadata like permissions, modification time.
CVE-2024-0450	CPython	MEDIUM	Missing detection of “quoted-overlap” ZIP bombs
CVE-2007-4559	CPython	CRITICAL	Path traversal during tar extraction
CVE-2025-8869	pip	MEDIUM	Fallback tar extraction filter bypass: symlink targets outside extraction directory. Only affects Python versions that don’t implement PEP 706.
CVE-2025-54368	uv	MEDIUM	ZIP confusion during extraction
GH-13877	PyPI	MEDIUM	Disallowing ZIP bombs
GH-18492	PyPI	MEDIUM	Disallowing ZIP confusion

WHY ARE ARCHIVE FORMATS SO COMPLICATED?

When you open a ZIP or tar archive, you see a list of files and directories, click ‘extract’, and there are all your files neatly as they appeared in your archive manager. Easy, right? Unfortunately, the underlying archive format is much more complicated than first impressions let on.

Archive formats are more accurately described as a series of instructions that occur to a filesystem than a neat set of files bundled together. This approach stems from a time when large archives needed to be split across multiple different files and drives (such as floppy disks). The “Central Directory” of ZIP archives specifies not only the absolute byte offset to a file, but also the “disk number” that the specific file is archived to. Nowadays, with terabyte microSD cards being readily available, these features are no longer as important day-to-day, but due to backwards compatibility, are still present in every ZIP and tar archive created.

Both ZIPs and tar archives support having additional file “entries” appended to the archive without removing the previous entries of the same name, and still produce a valid archive. This results in archives that can have duplicate entries with different content, permissions, or types (such as one entry being a link and another being a file). ZIP even supports deleting files within an archive by rewriting the Central Directory to remove the reference to a Local File. Implementations need to process these instructions consistently to be safe from confusion attacks.

Archive formats support many features beyond mapping file paths to data that further complicate implementations. Archives support many of the same features that filesystems do, such as creation and modification times, permissions, ownership, and links between different filesystem entries. Supporting these features, which are often platform-specific, means that implementations need to handle each in their respective platform-specific manner, leading to differences and complexity.

SUPPLY-CHAIN SECURITY AND ARCHIVES

Reproducible builds

Build reproducibility is a "set of software development practices that create an independently-verifiable path from source to binary code". Build reproducibility is one of the most potent detections (and thus, protections) that a software package has been compromised somewhere between the source and build phase. For example, if the xz-utils project had a fully reproducible build, the attack used by the malicious maintainer could have been detected if projects were checking the reproducibility of artifacts from the source code.

Software packages ideally should be byte-for-byte reproducible, but many features of archive formats make reproducing package archives difficult. Properties such as canonicalizing the order that file and directory entries appear within the archive and handling variable properties of the building system, such as user, the clock, tool versions, compression (stream names!), and more. It's quite a long list!

CPython's own build process contains the following options on the "tar" command to ensure the CPython source tarballs are byte-for-byte reproducible:

```
$ tar cf Python-$tag.tgz \
       --sort=name \
       --mtime=$(git log $tag -1 -pretty=%ct) \
       --clamp-mtime \
       --owner=0 --group=0 --numeric-owner \
       --pax-option \
       exthdr.name=%d/PaxHeaders/%f,delete=atime,delete=ctime \
       --mode=go+u,go-w \
       --use-compress-program "gzip --no-name -9" \
      ...
```

This long list of options is unwieldy, to say the least. If instead, there was a "--reproducible" option which enabled all these flags to achieve build reproducibility, that would be much more accessible and discoverable for users looking to create reproducible ZIP and tar archives.

Extraction filter escapes (“ZIP slip”)

When installing a software package, the installation program needs to extract the contents of the archive to the system before continuing with the package installation. This extraction process needs to be filtered to avoid modifying the system beyond what is intended.

Escaping this filtering is commonly done in one of two ways, either with “parent directory” path segments (e.g, “..”) or abusing links such as symbolic or hard links, either through extraction order, path collisions, or path truncations. When implementations aren’t filtering these attacks, this can result in malicious archives overwriting critical system files or binaries or data being read from confidential locations (“/etc/passwd”) before exfiltration through symlink attacks.

The trouble is that tar and ZIP archives are designed to be able to do these actions that can be abused by malicious archives. Threading the needle of “secure-by-default” is thus quite difficult for implementations, as they need to balance features that are used by real archives in the wild (such as parent directory segments in symlinks) with protecting users by default during installation of packages or routine extraction of archives.

Python addressed CVE-2007-4559 with “PEP 706”. PEP 706 adds “filtering” to tar archive extraction with three pre-defined filter modes: “fully_trusted”, “tar”, and “data”. There is a deprecation warning for using no defined filter when extracting an archive with the default changing to “data” in Python 3.14. Filtering allows for either rejecting or modifying archive members based on their contents, such as “defusing” absolute paths in members to be relative to the extraction directory. All installers are recommended to use filtering when extracting Python package archives to avoid attacks like path traversal.

Confusion and differential attacks

The complexity of the ZIP and tar archive standards means that implementations can vary from each other. These variations can be exploitable, especially if one implementation has variations that affect which files are “observed” within an archive during extraction. Using variations for which files are observed during extraction allows an attacker to smuggle files within an archive that aren’t detected, reviewed, or scanned, but then are installed on the target system upon installation.

This scenario was possible using CVE-2025-54368, where a package installer “uv” had a different extraction behavior than all other Python-based ZIP archive tools, such as PyPI, pip, inspector, and the Python standard library “zipfile” module. This meant that any tools used to scan a Python package for malware may not have detected smuggled files, but those files would be installed by uv.

Extraction denial-of-service (“ZIP bombs”)

Specially crafted ZIP archives, often called “ZIP bombs”, can be created to be only 10MB when compressed, but when extracted, result in hundreds of terabytes of data. This leads to resource exhaustion on the extracting system, as most systems aren’t prepared to handle this amount of data. This is possible because the ZIP and tar archive formats have separate entries for their “central directory” authoritative list of files within the archive, and “local file” entries which describe the file and its contents within the archive. This separation means that a single “file” can be referenced multiple, potentially thousands or millions of times, within a single ZIP file central directory, leading to a high and exploitable “compression ratio”.

A separate category of this vulnerability is creating archive files that, due to flaws in the implementation, never terminate the extraction process. This can be seen in CVE-2025-8194 where the “tarfile” module allowed tar member entry offsets to be negative, meaning that a member could loop the member parsing process into a previously visited state, resulting in an infinite loop during tar extraction.

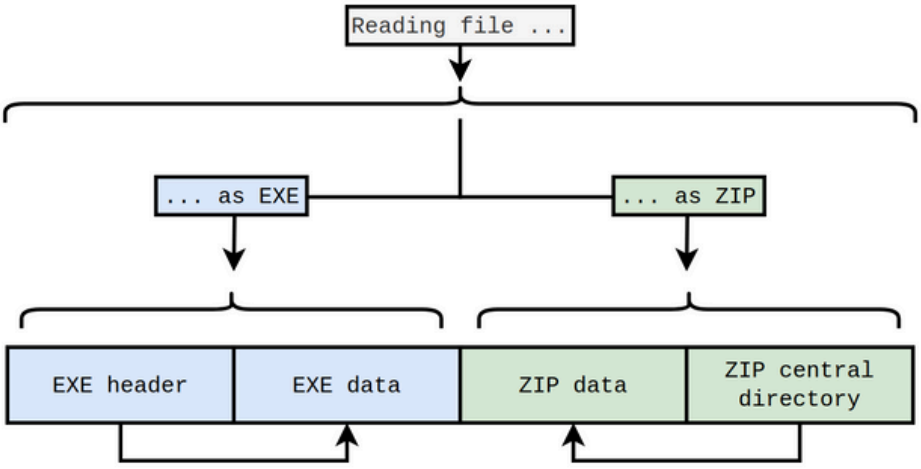
Denial of service vulnerabilities for archives aren’t a massive concern for installers, as the worst-case is that the installation fails or a system is rendered unusable until backup or restore occurs. However, for package repositories, a denial of service can interrupt service for users.

For package repositories, the easiest way to mitigate this vulnerability is to not extract the whole contents of an archive. Instead, most of the information required from an archive, such as checking the existence or contents of a metadata file or calculating a checksum, can be done without inflating the contents of the archive. To prevent ZIP bombs from having any impact on installers, package repositories can impose limits on the “compression ratio” and “uncompressed size” of archives. PyPI currently sets its ratio limit at 50x to allow for data-heavy packages to still be published.

Appended ZIPs (“zipapp”)

Unlike many file formats, ZIPs contain their “central directory” at the end rather than the beginning of the file. This quirk means that ZIPs can be read and recognized from the end of the file first, instead of the beginning. This means executables can have ZIPs appended to them to provide a “filesystem” that the executable can access by reading its own file path as a ZIP archive instead of an EXE.

This approach means the EXE and ZIP data will never overlap or interfere with each other. This feature exists in other programming languages, such as Turbo Pascal; for Python, the feature is called “zipapps”. Self-extracting ZIPs also use this technique, with a ZIP being appended to a ZIP extraction program.
As a program “opening” the appended ZIP, the file appears to either be an executable or a ZIP depending on which direction the file is read from:



To support this case, it must be possible to read and parse ZIP metadata without needing the absolute offset into the file from the beginning of the file, all operations need to happen securely and consistently from the end of the file.

The challenge comes with ZIP archives that implement the “ZIP64 extension” for supporting file sizes larger than can fit in a 32bit value. The ZIP64 extension adds multiple new ZIP records, including a ZIP64 variant of the “Central Directory” record and a new ZIP64 End of Central Directory Locator record. The End of ZIP64 Central Directory record allows an optional variable size “comment” field, and the End of Central Directory Locator record uses the absolute offset into the file as a location. These two features combined mean it’s impossible to accurately parse a ZIP64 archive using both features with “prepended data”. For this reason, to fix CVE-2025-8291 the zipfile module has to disallow using both features together in case of prepended data.

RECOMMENDATIONS AND FUTURE WORK

Restrict uncommon archive features

Packaging ecosystems should be specific in their use of archive formats like tar and ZIP. Disallowing features that aren't necessary, preventing archive features that could result in archive confusion attacks, and defining. For example, Python's "zipfile" module currently doesn't support multiple ZIP features, which are extremely uncommon today, such as ZIP archives that are spread across multiple "disks". Further changes from obscure features from "process by default" to "opt-in" may protect users against archive differentials.

Fuzzing archive and differential testing implementations

Archives formats are data exchange formats, and implementations of data exchange formats can benefit massively from fuzzing given quality seed corpora and structured fuzzers. In particular, fuzzing against "data" and "tar" filters introduced in PEP 706 to ensure their protection criteria, such as "rejects links," makes for easy-to-verify criteria for a fuzz target.

Differential testing, running the same inputs through two different implementations and seeing if the results match, is also an effective method of finding bugs in data exchange formats. ZipDiff, a new tool from the recently published paper "My ZIP is not your ZIP", which uses differential fuzzing of ZIP implementations and has already identified many differences between ZIP parser implementations, including Python's "zipfile" module. The implementation with the fewest differentials was "ronomon/zip," which is implemented explicitly to reject many sources of differentials at the expense of compatibility with all ZIP archives.

Canonical archive layouts and options for reproducibility

Defining a "canonical" archive layout within the specifications of package archives, along with tools or policy to push the ecosystem towards that canonical layout, can reduce reproducibility issues and make future improvements to restrict archive features possible.

Go's packaging ecosystem has a concept called a "DirHash," which is a pre-defined method of creating a checksum from the extracted file contents of an archive regardless of metadata, timestamps, and insertion order. "DirHash" is implemented by creating a SHA-256 checksum of a newline-delimited string of filenames and their respective SHA-256 checksums appended together. This shell script approximates the algorithm for "DirHash":

```
$ sha256sum $(find . -type f | sort) | sha256sum
```

This removes many system and implementation-specific differences of archives from affecting the final "checksum" of the package archive, making this mechanism more reliable to use compared to byte-for-byte comparisons. This checksum method doesn't account for links, empty directories, or archives that contain different contents depending on the extracting implementation.

Package repositories as an enforcement mechanism

Packaging ecosystems like Python's often have many tools with different organizational structures and roadmaps, and therefore cannot easily coordinate on ecosystem-wide challenges like archive format features. This is especially true when the issue arises from an ambiguity in a packaging standard, which can take months to fix and often requires public correspondence, meaning users are left to fend for themselves from exploits while standards are fixed.

Applying protections at the public package repository level would mean that would-be attackers can't exploit public information about a vulnerability at scale; instead would need to target private package repositories, which typically have fewer users or aren't available at all on the public internet, thus are less interesting and more difficult targets to exploit.

However, applying protections like rejecting archives from a large public package repository like PyPI is not without challenge. One of the biggest difficulties for developing open source package repositories is that maintainers don't know who or what is interacting with their systems. For this reason, there needs to be caution and care put into any decision that affects an entire ecosystem.

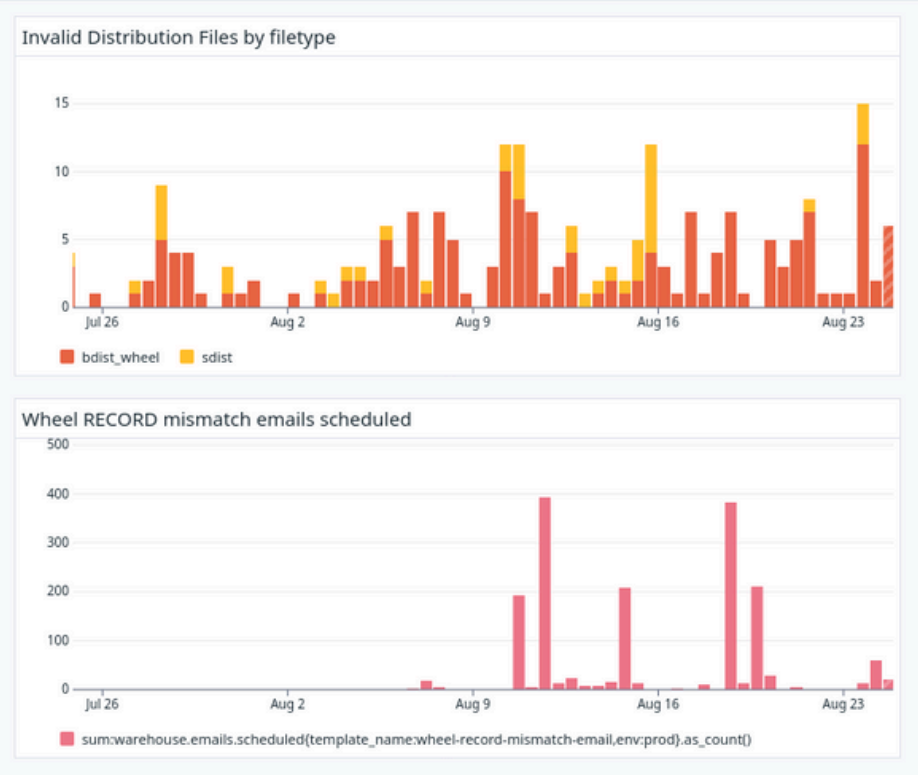
To evaluate the impact of these changes for rejecting ZIP and tar archives, the top 15,000 projects ordered by downloads were downloaded locally and tested for issues related to the ZIP format or RECORD mismatches. Additionally, a service that actively scans newly uploaded archives was created to estimate the occurrence across all PyPI projects, not only top projects. The results were that almost all published ZIP files had no issues:

Status	Number of Projects
No issues with ZIP format or RECORD	13,460
Missing file from RECORD	4
Mismatched RECORD and ZIP "Local File"	2
Duplicate files in ZIP "Local Files"	2

The remaining 1,532 projects didn't publish ZIPs, so they didn't have the above issues. With this data, we felt confident proceeding to implement all of the defensive measures protecting against ZIP confusion attacks.

The Python Package Index began emailing project maintainers who published packages with incorrect RECORD files inside packages. This was done after noticing that, despite the wheel package specification requiring installers to check the contents of the archive with the filenames listed in "RECORD", no popular installer was taking this step. After querying top projects on PyPI, this was found to be a relatively rare occurrence, so there was a short deprecation period to allow the few projects that did have issues time to correct without disruption. We opted for a deprecation instead of an outright error, as this wasn't as much a security issue as ZIP confusion, more an adherence to standards issue.

In the weeks since the changes have been deployed, there is an uptick in "invalid" distributions being rejected by PyPI, and there are emails being delivered to package maintainers about mismatched RECORD files.



We expect to see these numbers decrease as time goes on, ahead of the deprecation period expiring. As always, we continue to monitor the situation but haven't seen major disruptions to popular packages being published to PyPI as a result of these changes.

ACKNOWLEDGEMENTS

This white paper includes the work and security vulnerability reporting done by Caleb Brown (Google) and Tim Hatch (Netflix). Thanks to them for their responsible disclosure of vulnerabilities.