TRAIL OF BITS

Attestations: a new generation of signatures on PyPI

William Woodruff, Trail of Bits

introduction

hello!

- **William Woodruff**
  - open source group engineering director @ Trail of Bits
  - long-term OSS contributor (PyPI, PyPA) and maintainer (pip-audit, sigstore-python, zizmor)
  - @yossarian@infosec.exchange
- **Trail of Bits**
  - ~130 person R&D firm HQ'd in NYC
    - 80% remote
  - specialties: cryptography, compilers, program analysis research, "supply chain," OSS engineering, general high-assurance software development

thank-yous

- **this work was funded by Google's Open Source Security Team (GOSST)**
  - special thanks to Dustin Ingram and Hayden Blauzvern
- **additional thanks to Ee Durbin, Mike Fiedler, Donald Stufft, and Dustin Ingram for code and PEP reviews**
  - and for not skinning us when we accidentally broke PyPI for a few hours with a bad migration
- **multiple people on ToB's engineering team worked on this!**
  - special thanks to Facundo Tuesca and Alexis Challande for their design and engineering

Google Open Source Security Team

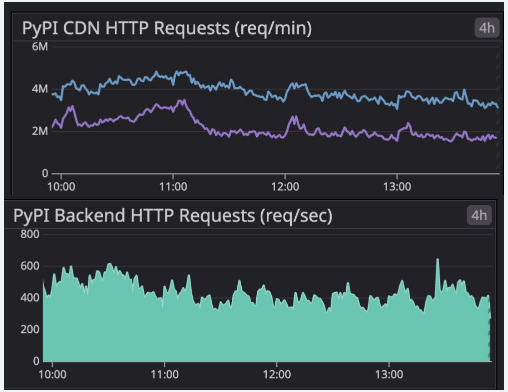
python™ Package Index

agenda

- **background: do we even have a problem?**
- **quick intro to digital signatures / code signing**
  - or: why the hell is this stuff so hard?
  - why (open source) signing schemes often fail in practice
- **why does any of this matter for PyPI?**
- **PEP 740 and attestations: rethinking signing from the ground up**
  - breaking the iron triangle of end-user signing, or: let's kill PGP
- **outcomes (so far)**
  - where PyPI and the rest of the ecosystem currently is
  - how ***you*** can produce and benefit from index-hosted attestations ***today***
- **looking forwards**
  - namely: bringing attestation *verification* closer to end users

background

do we even have a problem?



- **Python packaging *looks* very healthy!**
  - ~630K projects on PyPI, ~930K maintainers
  - 1B+ downloads per day, very heavily skewed towards the top 10K projects
- **many recent security improvements, both incremental and large:**
  - 2019/20: optional MFA (TOTP + WebAuthn) and scoped API tokens
  - 2022: mandatory MFA for critical projects
  - 2023: Trusted Publishing: automatic, self-scoping, self-expiring API tokens
  - 2024: mandatory MFA for all projects, mandatory API tokens for uploads
  - 2024: project quarantine + archival

what's even left to do?

background

“supply chain security”

- **open source package ecosystems (like PyPI) are a *double-edged sword:***
  - it’s all free and easy to use, so ***critical applications*** (gov’t, banks, power grids) ***depend on it***
  - it’s ***remote code execution as a service*** by design (which is why it’s so easy to use!)
- **security improvements like MFA make user-to-index protections stronger, but there are separate *user-to-user* and *index-to-user* risks:**
  - ***user-to-user risk***: insider/takeover risk (xz-utils), typo & slopsquatting
  - ***index-to-user risk***: attacker who compromises PyPI itself can modify any packages they want; undetectable by the (majority of) users who don’t do strict hash pinning

Here are the top 20 packages related specifically to Google APIs that would be useful for your grammar correction tool for Google Documents:

1. **google-api-python-client** - The official Python client library for accessing Google APIs
2. **google-auth** - Authentication library for Google APIs
3. **google-auth-oauthlib** - Tools for implementing OAuth 2.0 authentication flows
4. **google-auth-httplib2** - Google Auth integration with httplib2
5. **google-cloud-language** - Google's Natural Language API client with grammar analysis
6. **googleapis-common-protos** - Common protocol buffer types for Google APIs
7. **google-api-core** - Core functionality for Google API clients
8. **google-cloud-core** - Core libraries for Google Cloud client libraries
9. **google-resumable-media** - Utilities for Google Media Downloads and Resumable Uploads
10. **google-cloud-storage** - Client library for Google Cloud Storage (if storing documents)
11. **oauth2client** - Client library for OAuth 2.0 (legacy but still used)
12. **google-cloud-translate** - For translation services
13. **google-cloud-logging** - For logging and debugging your application
14. **gspread** - A wrapper library for Google Sheets API
15. **google-cloud-pubsub** - For implementing event-driven architectures
16. **google-cloud-error-reporting** - For tracking and reporting errors
17. **google-cloud-monitoring** - For monitoring your application's performance
18. **google-crc32c** - For data integrity checking
19. **protobuf** - Protocol Buffers library (used extensively by Google APIs)
20. **grpcio** - HTTP/2-based RPC framework used by many Google APIs

These packages cover authentication, document access, natural language processing, and various supporting utilities needed for building a robust Google Docs grammar correction application.

“supply chain security”

goal: improve the ***user-to-user*** and ***index-to-user*** security of Python packaging without degrading the things that make OSS awesome!

> ***subgoal***: it should be hard(er) for an attacker to impersonate or take over a project such that the average user of the project doesn’t/can’t detect the takeover

> ***subgoal***: it should be hard(er) for an attacker to compromise PyPI itself such that the average user doesn’t/can’t detect malicious changes to the projects they use

sounds like a problem for ***signatures***!

let's talk (public key) cryptography

- **public key cryptography: key materials come in *asymmetric* pairs**
  - the private half (the private key) is kept secret
  - the public half (the public key) is distributed to anybody who needs it
  - ***fundamentally asymmetric***: private half performs one operation (decrypting/signing) while public half performs the other (encrypting/verifying)
- **digital signature schemes are instantiations of PKC:**
  - private key holder is the ***signing party*** (Alice)
  - anybody with the public key can be a ***verifying party*** (Bob)
  - ***signing party*** does *SIGN*(K_{priv}, M) → S to produce S, a ***signature***
    - signing party can create signatures for any message they have access to
  - ***verifying party*** does *VERIFY*(K_{pub}, M) → P to produce P, a ***proof***
    - proof establishes ***authenticity*** (truly from Alice) and ***integrity*** (M is not modified)
    - verifying party can verify any signature they have the corresponding input for*

TL;DR: a true proof convinces Bob that Alice is the origin of M

let's talk code signing

- ***code signing*** **= digital signatures but for code**
  - M is a program instead of docs, emails, pictures of cats, etc.
  - ***integrity*** becomes "the program wasn't modified before I ran it"
  - ***authenticity*** becomes "the program was produced/conveyed by someone I trust"
- **If Bob trusts Alice (= Alice's key) to sign for package `foo`, then neither an attacker who compromises the project on PyPI nor PyPI itself can deliver a modified `foo` without Bob noticing!**

supply chain security solved??? we can all go home???

cryptography

not so fast

traditional signing schemes have *onerous assumptions:*

- ***secure distribution*: Alice can get her A_{pub} to Bob without Mallory replacing it with their own M_{pub}**
  - ***reality:*** requires an *additional* trusted party with its own keys/signatures, leading to regress
- ***timely revocation*: if Mallory steals A_{priv}, Alice must be able to revoke it such that Bob and others distrust it**
  - ***corollary***: Mallory *must not* be able to countermand a revocation of A_{pub}
  - ***reality***: revocations are difficult to operationalize and require infrastructure (revocation logs, trusted distributors) that ecosystems with *much greater resources* (= the web) have struggled to scale
- ***informed, expert users*: Alice knows how to generate a keypair correctly and keep it secure from Mallory for an indefinite period of time**
  - ***reality:*** most users ***don't understand cryptography, and should not have to***: like transport security on the web, authenticity of packages *should* be a latent property of the system itself

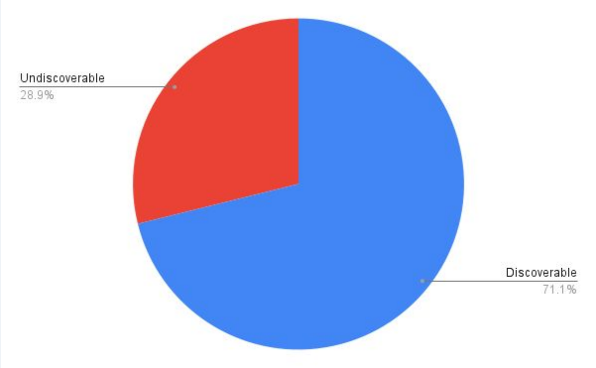
cryptography

(not so) ancient history: PGP on PyPI

PyPI's former support for PGP embodied these problems:

- ***promise:*** **users manually generate signing keys + upload `.asc` files**
  - **reality**: huge range of key types and sizes, making it hard to uniformly assert the quality/strength of a signature (plus poor key/cipher defaults in the ecosystem)
  - **reality**: users would upload all kinds of random stuff as `.asc`, including random garbage or keys instead of signatures
- ***promise***: **installers verify `.asc` signatures to establish authenticity and integrity!**
  - **reality:** punts key distribution/revocation to end users, who then have to navigate GPG arcane's CLI + deal with the PGP ecosystem's broken key distribution
  - **reality:** never widely adopted by installers due to limited signer-side adoption + onerous user requirements

outcome: PGP signatures never produced by the overwhelming majority of maintainers, and completely ignored by Python-level installers



✨ a new approach ✨

revisiting received wisdom in codesigning

- **assumption: both humans and computers verify through public keys**
    - **reality**: computers verify through keys, but humans only care about ***identities***
        - rephrased: I don't care ***which*** key Alice uses, as long as I can be convinced it ***is*** Alice's
- **assumption: humans are good at long-term secret management**
    - **reality:** humans are ***terrible*** at secret management!
        - entire corporate teams and industries struggle to do it; OSS maintainers shouldn't be thrown under the bus by security requirements they can't uphold
- **assumption: users understand basic cryptography**
    - **reality**: they might, but they ***shouldn't have to!***
        - similarly: users should not have to understand revocation, etc. to sign/verify safely
- **assumption: there's an acceptable "usability tax" for signatures**
    - **reality**: end-user usability is the only thing that matters in terms of driving user behavior
        - good security design therefore means *defaultable*, unintrusive design

a new approach

identities, not keys

we have a fundamental impedance mismatch:

- **keys are what crypto cares about (“the signature is valid for the key”)**
- **signing identities are what humans care about (“I trust Alice”)**

our goal is to collapse the distinction!

- **instead of weak identity claims (PGP: “I promise that I am `bill@msft.com`”), we want a *strong binding* between a *proven* signing identity and its key**
    - requires a new party: an ***identity provider*** that creates verifiable proofs of identity

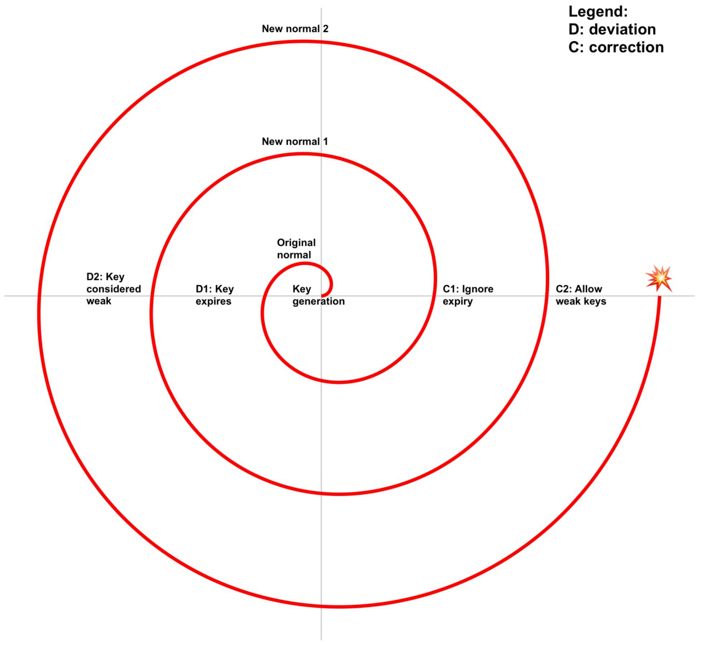
a new approach

in fact, no (long-lived) keys at all

managing long-lived signing keys is risky: keys are easy to accidentally leak, destroy, or *normalize* such that revoking/rotating them when necessary becomes operationally impossible

our goal is to build a scheme where deviance in key management *cannot be normalized!*

- **instead of long-lived keys, we want short-lived keys that *can't* be used deviantly**
    - requires a *reliable and highly available* mechanism for binding temporary keys to ***signing identities***



(with apologies to Diane Vaughan)

“bare” signatures aren’t all that great

traditional schemes sign over the bare bytes of the program, giving a proof like “the holder of K_{priv} is the authentic source of this M (program).”

but we *really* want to communicate a lot more than that: we want M to be *structured* such that, after verifying S, the verifier can *evaluate* M.

in other words, we want *attestations*:

- **machine-readable statements that can be verified like normal messages, but can be *additionally interpreted***
  - this enables ***domain binding***: we can sign ***not just*** the *contents* of a program, but also its domain: its distribution name, which index it’s going to, etc.

a new approach

bare signatures vs. attestations

index claims:

“the distribution named
`sampleproject-4.0.0.tar.gz`
corresponds to the digest
`sha256:01ba4719c80b…`”

🤔 **what parts of this does the signature *bind?***

🤔 **which parts can the index (or another party) *lie* about?**

a new approach

bare signatures vs. attestations

index claims:

> **"the distribution named**
> `sampleproject-4.0.0.tar.gz`
> **corresponds to the digest**
> `sha256:01ba4719c80b…`**"**

***bare signature*:**

M = H(read("sampleproject-4.0.0.tar.gz"))

S = SIGN(K_{priv}, M)

S binds the digest *and only* the digest!

the index (or another party) can *mix-and-match* valid signatures with distribution names.

bare signatures vs. attestations

index claims:

"the distribution **named** `sampleproject-4.0.0.tar.gz` corresponds to **the digest** `sha256:01ba4719c80b…`"

attestation:

M = H("sampleproject-4.0.0.tar.gz:" + read("sampleproject-4.0.0.tar.gz"))

S = SIGN(K_{priv}, M)

S binds both the distribution name ***and*** the digest!

the index ***can't*** mix-and-match the signature with the distribution name…

…but there are always other domains (e.g. "this package belongs to a *specific* index")

✨PEP 740✨

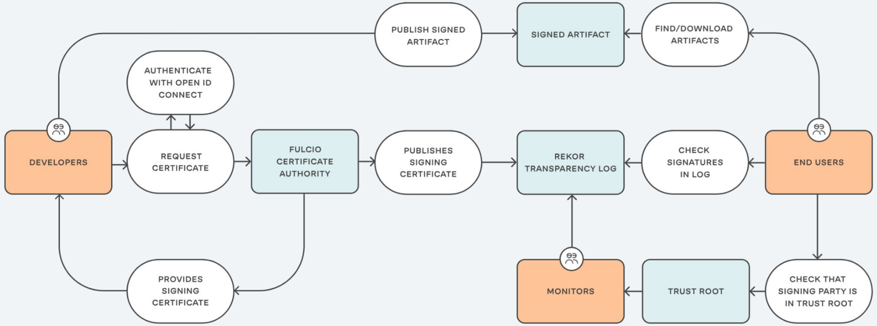
PEP 740 is…

…built on Sigstore

Sigstore provides the parties and mechanisms that enable our approach:

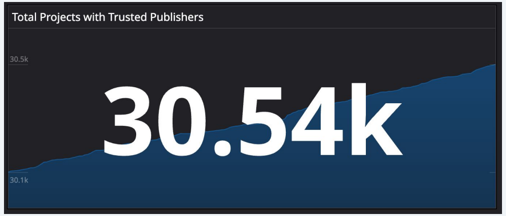
- ***identities over keys*** **come from Fulcio, Sigstore's OIDC PKI**
  - Fulcio takes a public key + OIDC credential, verifies it, and issues a ***certificate***
    - the certificate ***binds*** the identity in the OIDC cred to a public key
    - this binding is ***verifiable and auditable*** via Certificate Transparency!
- ***keyless signing*** **comes from short-lived, Fulcio issued certificates**
  - certificates are only valid for 10 minutes at a time!

…but which identities to trust?



PEP 740 is…

…built on Trusted Publishing



- **Trusted Publishing is PyPI's "no-credential" authentication scheme**
    - built on top of PyPI's API token scheme, but uses OIDC to allow a verifiable identity (e.g. a workflow in a GitHub repository) to request short-lived publishing tokens
- **supports GitHub, GitLab, Google Cloud, and ActiveState as publishers**
    - widely adopted (>30K projects, hundreds of thousands of releases/files)
    - used by default for GitHub!
- **Trusted Publishing *establishes a machine identity!***
    - we can reuse this identity for signing!

Manage current publishers

Publisher	Details	
GitHub	**Repository:** pypa/pip-audit **Workflow:** release.yml **Environment name:** release	Remove

```yaml
jobs:
  pypi-publish:
    name: Upload release to PyPI
    runs-on: ubuntu-latest
    permissions:
      id-token: write
    steps:
    - name: Publish package distributions to PyPI
      uses: pypa/gh-action-pypi-publish@release/v1
```

PEP 740 is…

…built on in-toto

in-toto is a ***framework*** for defining machine-readable attestations

- solves the "bare signature" problem for us

PEP 740 defines a "publish" attestation using in-toto, which we then sign over:

```json
{
  "_type": "https://in-toto.io/Statement/v1",
  "subject": [
    {
      "name": "rfc8785-0.1.2-py3-none-any.whl",
      "digest": {
        "sha256": "c4e92e9ecc828bef2aa7dba1de8ac983511f7532a0df11c770d39099a25cf201"
      }
    }
  ],
  "predicateType": "https://docs.pypi.org/attestations/publish/v1",
  "predicate": null
}
```

PEP 740

tying it all together

- Sigstore gives us misuse-resistant, identity based signing
- in-toto gives us structured machine-readable attestations
- Trusted Publishing gives us a ***pre-established trust relationship***
  - one that's ***already used by default*** in major publishing workflows!
  - one that the ***index itself can verify*** as an upload criteria!

combined, these *dispel* the traditional onerous assumptions in signing:

- ***secure distribution*** via strong identity binding, plus mandatory transparency (Certificate Transparency and "signature transparency" in Rekor)
- ***timely revocation*** obviated by keys that expire long before conventionally accepted revocation periods (~24 hours for the Web PKI)
- ***users*** get signing by default, without having to learn cryptography!
- ***index verification*** establishes a signal and quality floor for the entire ecosystem

demo time!

where are we ***now***?

quick recap: where we were with PyCon 2023…

ergonomic codesigning for the Python ecosystem

William Woodruff

PyCon 2023 | Ergonomic codesigning for the Python ecosystem 2

sigstore

can we do better? Yes, with Sigstore!

Sigstore is…

- **…a PKI* ecosystem (X.509 CA, RFC 6962 CT)**
  - "Let's Encrypt but for code signing"
- **…a framework for *binding keys to public identities***
  - Leveraging OpenID Connect to communicate with well-known identity providers (IdPs) like Google, GitHub, etc.
- **…a client ecosystem**
  - Per-language/ecosystem bindings, like for Python!
- **…a Linux Foundation project**
  - Developed and maintained by members of the OSS community

PyCon 2023 | Ergonomic codesigning for the Python ecosystem 13

sigstore for python

Sigstore for Python: where we are

- **we have a mature Sigstore impl. for Python (`sigstore-python`) that can be used to sign anything (including Python distributions)**
- **we have a GitHub Action (`gh-action-sigstore-python`) that makes signing *completely painless* on CI**
- **CPython itself is signing its official releases with Sigstore, using each release maintainers' email identity:**

https://www.python.org/download/sigstore/

PyCon 2023 | Ergonomic codesigning for the Python ecosystem 18

sigstore for python

Sigstore for Python: where we want to be

- **PyPI should allow uploading of Sigstore bundles next to their associated release files**
  - …including with a pairing/TOFU* scheme against the package's pre-existing unauthenticated metadata
  - Related PEPs: 694, 691, an unwritten one for the TOFU scheme
- **`pip` should (optionally, at first) verify Sigstore bundles during download/install**
  - 2FA for critical projects rollout as a template: mandate signatures for the top N projects

PyCon 2023 | Ergonomic codesigning for the Python ecosystem 19

this talk!!!!!

where are we ***now***?

- **PEP 740 is fully implemented on PyPI!**
  - MVP completed last November, features being added continuously since
- **attestations are *enabled by default* in gh-action-pypi-publish**
  - (see top right)
  - available but not default ***yet***: GitLab and Google Cloud Build
- **PyPI *verifies* and presents attestation metadata on the file details page for each uploaded distribution**
  - (see bottom right)
  - includes verifiable source provenance + auditable transparency log entries!!!

```yaml
jobs: pypi-publish
  permissions:
    id-token: write
  steps:
    - uses: pypa/gh-action-pypi-publish@release/v1
```

Provenance

The following attestation bundles were made for `sampleproject-4.0.0.tar.gz`:

Publisher: `release.yml` on pypa/sampleproject

Attestations:

Values shown here reflect the state when the release was signed and may no longer be current.

- Statement:
  - Statement type: `https://in-toto.io/Statement/v1`
  - Predicate type: `https://docs.pypi.org/attestations/publish/v1`
  - Subject name: `sampleproject-4.0.0.tar.gz`
  - Subject digest: `0ace7980f82c5815ede4cd7bf9f6693684cec2ae47b9b7ade9add533b8627c6b`
  - Sigstore transparency entry: 147137139
  - Sigstore integration time: Nov 6, 2024, 5:37:07 PM

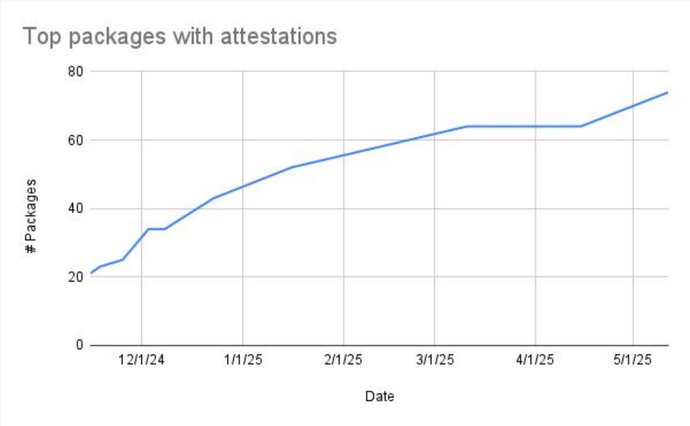
Source repository:
- Permalink: `pypa/sampleproject@621e4974ca25ce531773def586ba3ed8e736b3fc`
- Branch / Tag: `refs/heads/main`
- Owner: https://github.com/pypa
- Access: `public`

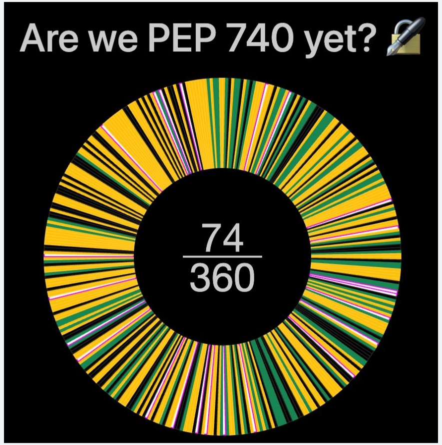
Publication detail:
- Token Issuer: `https://token.actions.githubusercontent.com`
- Runner Environment: `github-hosted`
- Publication workflow: `release.yml@621e4974ca25ce531773def586ba3ed8e736b3fc`
- Trigger Event: `push`

where are we ***now***?

show me the numbers

- **14% of all packages uploaded since 2024-11 have attestations!**
  - overall trend is accelerating, as people both roll onto Trusted Publishing and upgrade their workflows to ones that sign by default
- **~21% of all top-downloaded packages have attestations!**
- **for contrast, PGP signing on PyPI:**
  - had ~4% adoption when encouraged (2016), as low as 0.3% in 2023 (prior to deprecation announcement)
  - overcounts invalid, expired, etc. signatures





where are we ***now***?

auditing and discoverability wins

enforced transparency makes incident response easier!

Ultralytics attack in December:

- **tlog complements server-side logs, giving us additional timeline details**
- **tlog provides exact build/source state metadata, telling us the GitHub repo's exact state at the time of exploitation**

Supply-chain attack analysis: Ultralytics

Last week, the Python project "ultralytics" suffered a supply-chain attack through a compromise of the projects' GitHub Actions workflows and subsequently its PyPI API token. No security flaw in PyPI was used to execute this attack. Versions 8.3.41, 8.3.42, 8.3.45, and 8.3.46 were affected and have been removed from PyPI.

where are we ***now***?

manual end-user verification 👎

signing ✅, index-side verification ✅, *manual* end-user verification…💀

brave users can experiment with `pypi-attestations` **(library or CLI):**

```
arbet:~ william$ uvx --prerelease=allow pypi-attestations \
  verify pypi \
  --repository https://github.com/pypa/sampleproject \
  pypi:sampleproject-4.0.0-py3-none-any.whl
OK: sampleproject-4.0.0-py3-none-any.whl
```

where ***next***?

where ***next***?

default end-user verification!

hard truth: the impact of *any* signing scheme is marginal without verifying parties

- **ideally, lots of them**
- **in our case we have PyPI as a “baseline” verifier, but we can do a *lot* better!**

goal: get verification into users’ hands by *default*

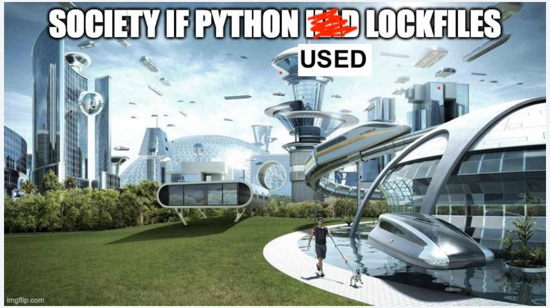
default verification: TOFU with PEP 751 lockfiles!

PEP 751 standardizes lockfiles for Python!

- **accepted just this March!**
- **thank you Brett!**

PEP 751 includes metadata for PEP 740

- **allows us to lock attestation *identities***




```
[[packages]]
name = 'attrs'
version = '25.1.0'
requires-python = '>=3.8'
wheels = [
  {name = 'attrs-25.1.0-py3-none-any.whl',
]
[[packages.attestation-identities]]
environment = 'release-pypi'
kind = 'GitHub'
repository = 'python-attrs/attrs'
workflow = 'pypi-package.yml'
```

where ***next***?

TOFU with PEP 751 lockfiles!

PEP 751 gives us the building blocks for a TOFU scheme:

- **users do `pip lock` or similar to initialize their lockfile**
  - initialization ideally (but not yet) includes `[[project.attestation-identities]]`
- **on upgrades, `pylock.toml` is consulted for locked identities**
- **upgrade candidates are verified against locked identities**
  - allows us to fail if an attacker tries to ***substitute*** their identity!
  - allows us to fail if an attacker tries to ***downgrade*** to un-attested releases!
- **satisfies the goal of *default* verification without exposing users to cryptographic building blocks**
  - …at the cost of a TOFU phase

(a bit) further out: monitoring!

- **TOFU is good, but we can do better!**
- **enforced transparency for attestations means trivial *monitoring*:**
  - maintainers can monitor their ***signing identities*** for unauthorized activity
  - community can monitor ***package identities*** for “ghost” releases
- **goal: something like Cert Spotter (TLS) or GopherWatch (Go), but for Python!**

GopherWatch

Keep tabs on Go modules.

Subscribe to Go module paths and receive an email when a new module/version is published through the Go module proxy.

How does it work?

In Go, you use "go get" to download Go modules to use as a dependency. Retrieving a module is done through the Go module proxy, which adds all module versions to the Go checksum database, a transparency log: A signed, append-only public log containing module versions along with a hash of their contents, providing high assurance that everyone requesting a module gets the same code. It is just like certificate transparency logs for TLS certificates.

GopherWatch follows the modules/versions appended to the Go sum database. You can subscribe to modules. GopherWatch sends you an email whenever a new matching module/version appears in the append-only log.

take-aways

take-aways

defaults matter: PEP 740 works (vs. historical approaches to signing) because it notches into existing workflows ***by default***

- corollary: defaults ***need*** to be unintrusive, ***especially*** in long-tail ecosystems like Python!

getting ***to*** a default-ready state took *years* of incremental changes:

API tokens, Trusted Publishing for machine identities, Sigstore's development, etc. all needed to happen *before* PEP 740 made sense

take-aways

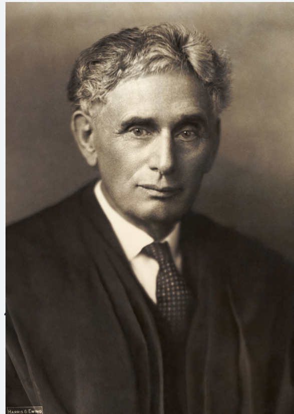
trust is unavoidable: PEP 740 avoids the key/identity distribution problem by ***shuffling*** trust, not ***removing it***

- corollary: if we can't ***remove*** trust, we need technologies that allow us to ***substantiate*** trust instead of accepting it blindly!

PEP 740 adds trust in new parties: OIDC providers, Sigstore's PKI, etc…

…but these parties are ***made accountable*** through transparency logs!

or bluntly: we cribbed the Web PKI's model for making CAs accountable because it's ***proven to work***, unlike e.g. webs of trust



take-aways

signing is the "easy" part: there's a long way to go in terms of ***default end-user verification***

- index verification is good, but ***not good enough***

PEP 751 gets us closer, but it's a long road:

- lockfile generators need to ***embed*** identities (for TOFU)
- installers/updaters need to ***consult*** locked identities
- users need to ***understand*** when and why (not) to trust identity changes
  - this last part is hard, but hopefully not as hard as opaque key IDs!

contact 👉

slides 👉

thank you!