TRAIL OF BITS

Securing your Package Ecosystem with Trusted Publishing

William Woodruff

Hello!

- **William Woodruff (william@trailofbits.com)**
  - open source group engineering director @ trail of bits
  - long-term OSS contributor (Homebrew, LLVM, Python) and maintainer (pip-audit, sigstore-python)
  - @yossarian@infosec.exchange
- **Trail of Bits**
  - ~150 person cybersecurity engineering and auditing consultancy
  - specialities: cryptography, compilers, program analysis research, “supply chain”, OSS package management, general high assurance software development

Some thank-yous

- **This work wouldn't have happened without Dustin Ingram and the GOSST team's vision for improving PyPI's security!**
- **The other maintainers of PyPI (Donald, Ee, Mike) all reviewed this work or otherwise made it possible**
- **Other members of PyPA for being early testers and ensuring a smooth rollout**
  - Special thanks to Sviatoslav Sydorenko
- **PyCA maintainers (Paul and Alex) for being guinea pigs and providing early feedback on usability**


This talk

Act 1: PyPI and Trusted Publishing

- **Quick intro to PyPI and Python package publishing**
- **Making PyPI publishing more secure with Trusted Publishing**
- **Outcomes so far**

Act 2: Trusted Publishing, 💖*for you*💖

- **Explaining and communicating to end users (package uploaders)**
- **Implementation tips and tricks (from our experience)**
- **Knock-on effects and benefits**

Background: PyPI

- **Pronounced 🥧-□-👁️**
- **The primary package index for the Python ecosystem**
  - ~500K projects, ~5M releases, ~9M files, ~750K users (i.e. packagers)
  - ~20 **billion** downloads per month (August 2023)
- **Rewritten in ~2017, security features added since**
  - 2018/19: API tokens, TOTP and WebAuthn, security event logging for users and projects
  - 2019/20: Malware scanning
  - 2020/21: Vulnerability feeds, GitHub secret scanning integration
  - 2022/23: Trusted publishing (you are here!)

Publishing on PyPI: then and ~now

- **< 2019: username/password authentication for uploads**
  - 👎 *No credential separation*: same user/pass could **modify all projects** under the same user!
    - As well as log into PyPI and do normal account admin things
  - 👎 *No straightforward revocation:* compromise means full account recovery needed!
  - 👎 *No security events:* attacker who steals creds can remain (relatively) stealthy!
- **>= 2019: Macaroon-based API tokens**
  - 👍 Configurable scopes (per-user, per-project)
    - Per-user means “all projects,” not “can modify the user’s profile”!
  - 👍 Integrated into security events + GitHub secret scanning
  - 👍 Backwards compatible with user/pass auth (no downstream tooling changes needed!)

API tokens are a major improvement; can we do even better?

What does “doing better than API tokens” look like?

For security:

- **API token configuration is error prone (fatigue & overscoping)**
  - We want something that doesn’t need to be manually configured or copy-pasted into a CI/CD system!
- **API token revocation is manual and brittle**
  - We want something that doesn’t *need* revocation!
  - *…without* requiring constant rotation or other fatigue-inducing constraints!
- **API tokens still “fail open”**
  - An attacker with a stolen token can wait indefinitely to use it
    - Murphy’s law but for security: an empowered attacker will always strike at the worst time
  - User-scoped tokens typically grow in scope over time
  - We want fewer passive attack vectors and the smallest viable credential scope!

What does “doing better than API tokens” look like?

For usability:

- **API tokens are associated with users**
  - Even when scoped to projects!
  - This causes logistical pains when a project/repository changes owners
- **Avoiding “chicken-and-egg” problems**
  - API tokens are immutable + can’t be scoped for a nonexistent project
  - To securely scope a token, users currently have to less securely create the project they’re trying to secure! Not great!

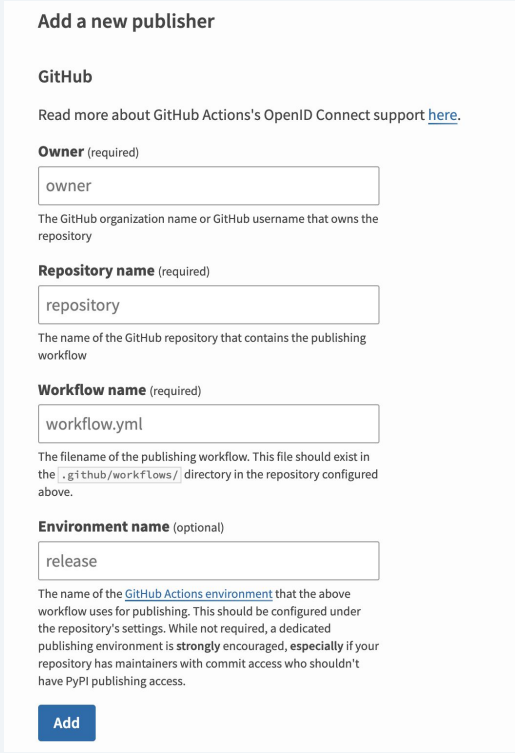
Doing better with ✨OpenID Connect✨

- **CI/CD providers like GitHub support *machine identities* through OIDC**
  - These credentials are strongly bound to the repository + workflow that made them
  - Can be verified by any third-party service using OIDC Discovery!
- **OIDC credentials are short-lived + scoped to an intended audience**
  - Fewer compromise opportunities + no domain contamination!
  - Service B won't accept credentials made for Service A
- **Already widely applied to other services (GH ←→ AWS, GCP, etc.)**
  - Why not PyPI too?

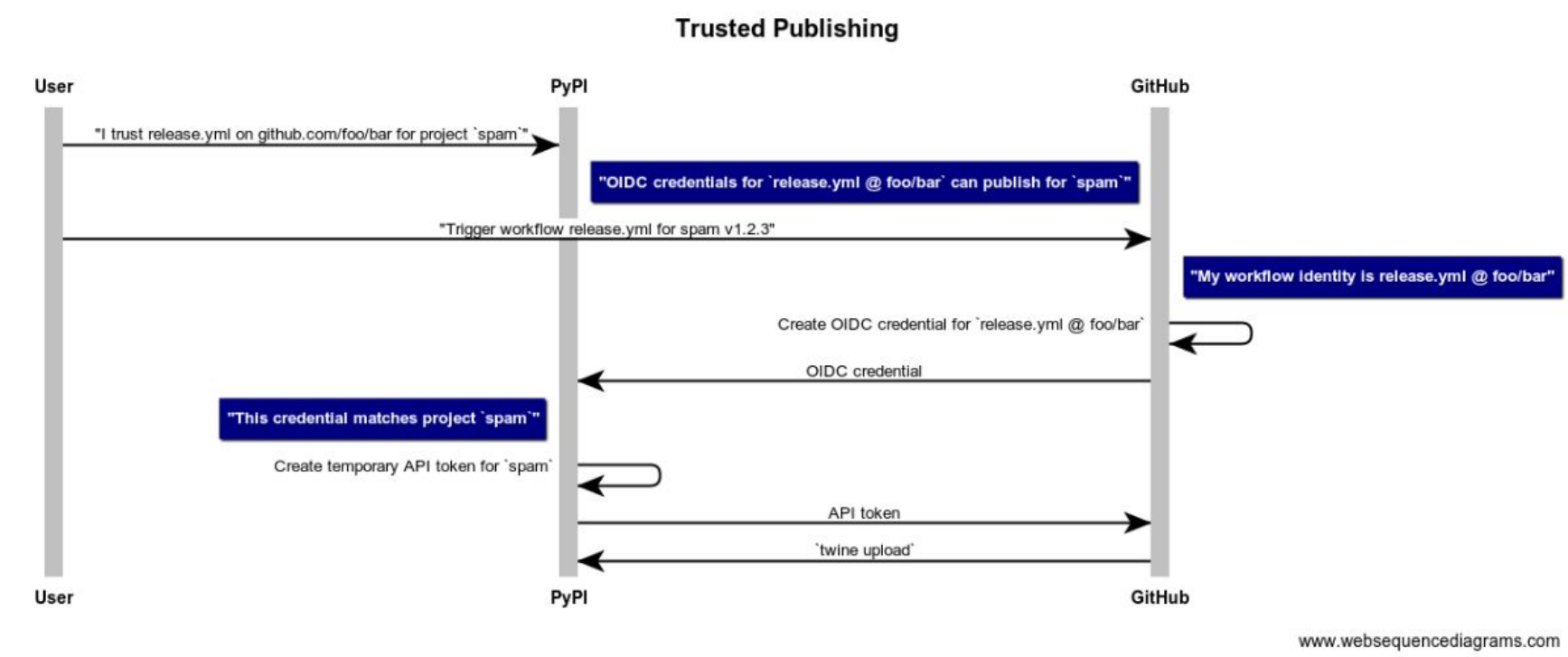
OpenID Connect for PyPI publishing

The rough idea:

- **Ahead of time: Users configure a *trust relationship* between a specific OIDC provider (e.g. GitHub) and their PyPI project**
  - The trust relationship itself isn't secret, so no potential leaks here!
  - For GitHub: user/repo slug, workflow name (e.g. `release.yml`), optional environment name
  - We call this relationship the "trusted publisher"
- **During publishing: OIDC provider creates an OIDC credential**
  - PyPI accepts that credential, verifies it, and *exchanges* it for a short-lived PyPI API token scoped for the project
  - Package publishing (e.g. through `twine`) continues as normal, none the wiser!



Trusted publishing: the bird's-eye view

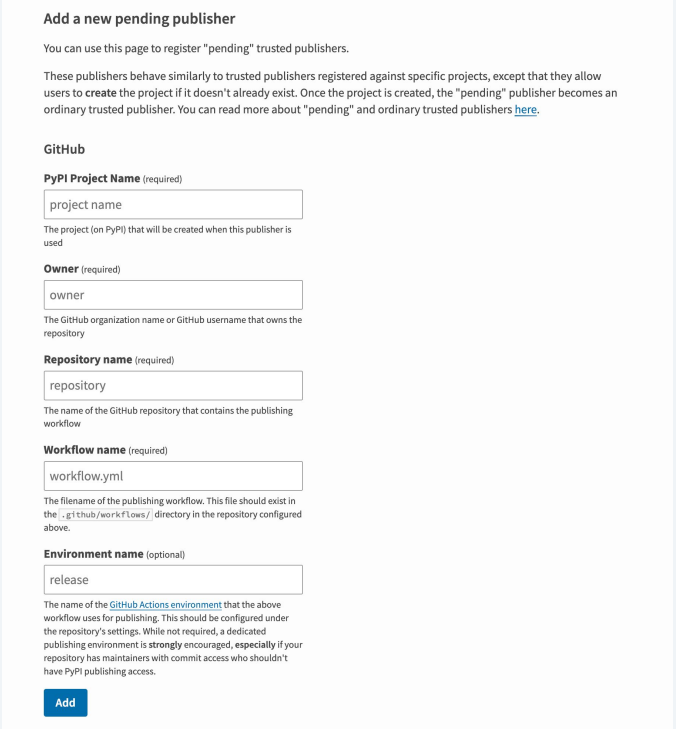


What about nonexistent projects?

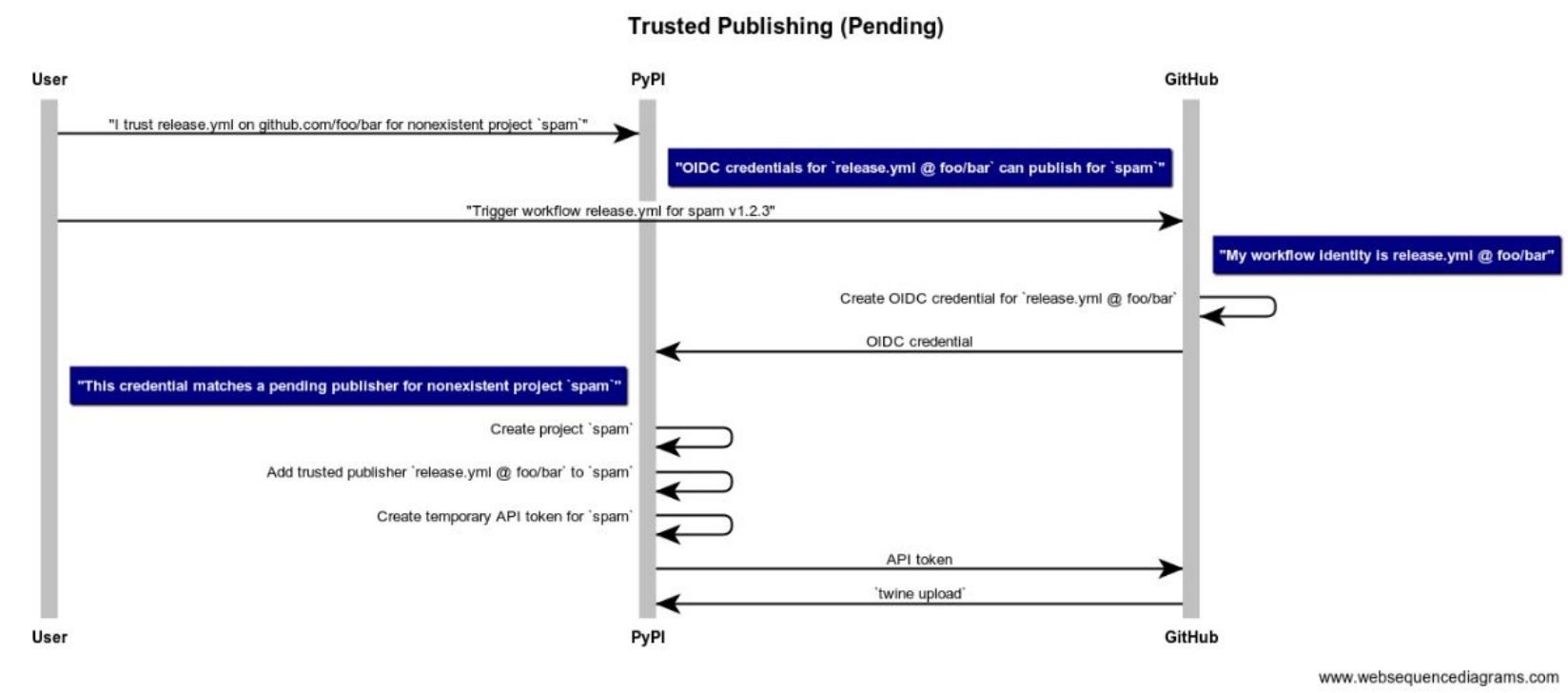
This first approach doesn't solve the "chicken-and-egg" problem with API tokens: the project still needs to exist to register a trusted publisher to it!

We solve this with "pending publishers": registered similarly to a trusted publisher, but associated with a user instead.

- **Contains the name of the project that will be created**
- **On first use, the project is created (in an empty state) and the "pending" publisher is *reified* into a full "trusted publisher"**



Pending publishers: the bird's-eye view



It works!

With this scheme, we achieve all of our security goals:

- ✅ **All credentials are temporary and self-expiring**
- ✅ **All credentials are minimally scoped (no user scopes)**
- ✅ **Users only perform configuration once (initial trusted setup)**
- ✅ **All configuration is over public information (no private metadata)**
- ✅ **No more chicken-and-egg (“pending” publishers transition seamlessly to full publishers once used)**

We also solve supply chain problems in the process:

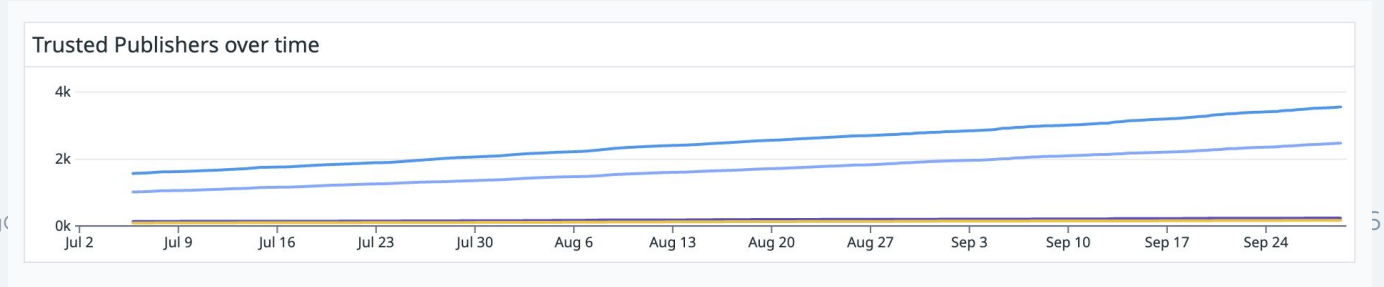
- ✅ **Flattening of state: the source repository itself becomes the “ground truth”**
- ✅ **Maintenance transitions: projects can transition maintainers without playing “who owns the credential”**

Outcomes (so far)

2023-04-20: trusted publishing becomes generally available on PyPI

How are we doing, ~6 months later?

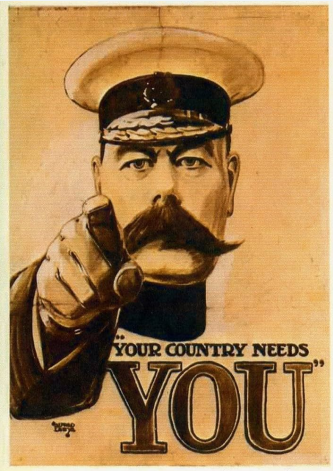
- **~4380 total projects configured to use trusted publishing**
  - ~3080 (~70%) have published at least one release using a trusted publisher
  - Takeaway: PyPI users are proactively configuring trusted publishing!
- **Critical projects: 272 configured to use trusted publishing**
  - ~5.2% of all critical projects have a trusted publisher configured
  - 198 (~73%) have published at least one release using a trusted publisher
- **Growth has been roughly linear:**



Zooming in on critical projects

~5.2% doesn't sound great, until broken down by downloads!

- **PyPI has ~6B downloads/week**
- **>25% of the top 25 projects are using trusted publishers:**
  - urllib3, charset-normalizer, certifi, wheel, cryptography, click
- **Adoption has generally skewed towards more popular packages, and PyPI downloads follow a power-law distribution!**
  - Even top packages show a power law: #1 has 9x as many downloads as #20
- **Takeaway: each critical/popular project that switches helps improve security across its entire dependency graph!**
  - If **you** are the maintainer of a popular Python project, then consider switching!



part 2: trusted publishing, 💖*for you*💖

tips for other possible implementations

Tip #1: the data model is unintuitive

"One trusted publisher per package, how hard could it be?"

Realities:

- **Multiple logical projects live under the same logical publisher**
  - Example: GitHub monorepo with multiple PyPI projects
- **Multiple logical publishers are responsible for a single project**
  - Example: Single PyPI project with multiple arch-specific publishing workflows
    - Users ideally wouldn't do this, but we want to encourage adoption!

Conclusion: trusted publishing is actually many-many; this may have surprising complexity implications for the ecosystem you're adding it to!

```python
class OIDCPublisherProjectAssociation(db.Model):
    __tablename__ = "oidc_publisher_project_association"

    oidc_publisher_id = Column(
        UUID(as_uuid=True),
        ForeignKey("oidc_publishers.id"),
        nullable=False,
        primary_key=True,
    )
    project_id = Column(
        UUID(as_uuid=True), ForeignKey("projects.id"), nullable=False, primary_key=True
    )
```

Tips from PyPI's implementation

Tip #2: OIDC is *very* narrowly standardized

"They're just JWTs under the hood, how different could they be?"

Realities:
Individual providers have wide latitude in claim availability/format

- **Only basic things can be assumed to be universal: `iss`, `exp`, `aud`, etc.**
- **Each new provider needs to be carefully inspected to determine *which parts* of the OIDC credential constitute sufficient trusted metadata**
  - This requires in-depth knowledge of the provider's internals/behavior, e.g. which users are entitled to run GitHub workflows within a particular repository!

Conclusions:

- **Adding new trusted publisher providers (GitLab, etc.) is time intensive; ecosystems should prioritize the providers they see used the most (GitHub for PyPI)**
- **Supporting multiple providers = more data model complexity!**

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NO
as described in **RFC 2119** [RFC2119].

```python
class OIDCPublisherMixin:
    """
    A mixin for common functionality between all OIDC publishers, including
    "pending" publishers that don't correspond to an extant project yet.
    """

    # Each hierarchy of OIDC publishers (both `OIDCPublisher` and
    # `PendingOIDCPublisher`) use a `discriminator` column for model
    # polymorphism, but the two are not mutually polymorphic at the DB level.
    discriminator = Column(String)

    # A map of claim names to "check" functions, each of which
    # has the signature `check(ground-truth, signed-claim, all-signed-claims) -> bool`.
    __required_verifiable_claims__: dict[str, CheckClaimCallable[Any]] = dict()

    # Simlar to __verificable_claims__, but these claims are optional
    __optional_verifiable_claims__: dict[str, CheckClaimCallable[Any]] = dict()
```

```python
class OIDCPublisher(OIDCPublisherMixin, db.Model):
    __tablename__ = "oidc_publishers"

    projects = orm.relationship(
        Project,
        secondary=OIDCPublisherProjectAssociation.__table__,  # type: ignore
        backref="oidc_publishers",
    )
    macaroons = orm.relationship(Macaroon, cascade="all, delete-orphan", lazy=True)

    __mapper_args__ = {
        "polymorphic_identity": "oidc_publishers",
        "polymorphic_on": OIDCPublisherMixin.discriminator,
    }
```


Tips from PyPI's implementation

Tip #2.5: OIDC varies wildly *within* IdPs

"Every trusted publisher through e.g. GitHub should look basically the same"

Realities:

- **OIDC identities vary wildly even within a provider: GitHub has special claims for reusable workflows, claims for different CI event types, etc.**
  - Providers like to change their claims without telling anyone!
- **Differences between these claims can't be paved over without (1) excluding some users or (2) ignoring some claims that might be important!**

Conclusion:

- **Supporting every possible configuration of a trusted publisher is hard + trying to do so opens up a lot of potential logic errors!**

```python
@staticmethod
def __lookup_all__(klass, signed_claims: SignedClaims) -> Query | None:
    # This lookup requires the environment claim to be present;
    # if it isn't, bail out early.
    if not (environment := signed_claims.get("environment")):
        return None

    repository = signed_claims["repository"]
    repository_owner, repository_name = repository.split("/", 1)
    workflow_prefix = f"{repository}/.github/workflows/"
    workflow_ref = signed_claims["job_workflow_ref"].removeprefix(workflow_prefix)

    return (
        Query(klass)
        .filter_by(
            repository_name=repository_name,
            repository_owner=repository_owner,
            repository_owner_id=signed_claims["repository_owner_id"],
            environment=environment.lower(),
        )
        .filter(
            literal(workflow_ref).like(func.concat(klass.workflow_filename, "%"))
        )
    )
```

```python
@staticmethod
def __lookup_no_environment__(klass, signed_claims: SignedClaims) -> Query | None:
    repository = signed_claims["repository"]
    repository_owner, repository_name = repository.split("/", 1)
    workflow_prefix = f"{repository}/.github/workflows/"
    workflow_ref = signed_claims["job_workflow_ref"].removeprefix(workflow_prefix)

    return (
        Query(klass)
        .filter_by(
            repository_name=repository_name,
            repository_owner=repository_owner,
            repository_owner_id=signed_claims["repository_owner_id"],
            environment=None,
        )
        .filter(
            literal(workflow_ref).like(func.concat(klass.workflow_filename, "%"))
        )
    )
```

```python
__lookup_strategies__ = [
    __lookup_all__,
    __lookup_no_environment__,
]
```

terrible!

Tip #3: OIDC tokens are not API tokens

“An OIDC credential is basically an API token; I don’t need to do an exchange”

Reality: you *can* do this, but…

- **OIDC credentials aren’t plugged into your preexisting AuthN/Z or permissions/scopes; you’ll end up re-implementing a bunch of what you already have (and reimplementation means more bugs)**
- **OIDC credentials are ~~chonky~~ contain all kinds of stuff you might not want to hold onto for prolonged periods (user emails, other potential PII)**
- **IdPs can change expiration and other policies without notice; creating your own temporary token makes you resilient to these changes!**

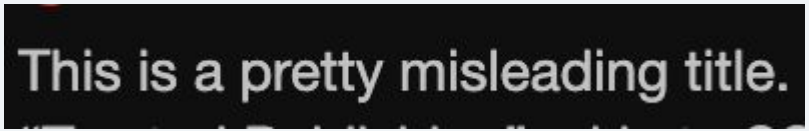
Conclusion: Performing token exchange minimizes the amount of novel code needed; reduces potential sources of PII; offers additional resilience against IdP changes.

The name is inaccurate and even misleading i

Tip #4: Words are hard

This is a great feature, but its name is going to confuse people.

Definitely a step in the right direction, but not what I thought it was going to be given the name '



. The name of this program is misleading.

Have to say this is a REALLY misleading name. '

Tip #5: It's all worth it!

OIDC is complicated; trusted publishing's model on top of it even more so.

But the *user experience and security gains make it worth it*:

- **User feedback (once they understand it) is overwhelmingly positive**
  - High demand for more trust relationships (e.g. reusable GitHub workflows)
  - High demand for additional IdPs (GitLab, CircleCI, GCP, BuildKite)
  - High demand for APIs and UI elements that expose trusted publishing status/metadata
- **Adoption by critical projects is steadily advancing**
  - …and is having a disproportionate impact on the entire ecosystem's security

The future

What comes next?

The same building blocks that give us trusted publishing (OIDC, machine identities) are *also* the building blocks for build provenance and code signing!

The goal: ecosystems that support trusted publishing should find it relatively easy to enable Sigstore for codesigning.

Event

File added to version 2.6.1
Filename: `pip_audit-2.6.1.tar.gz`
Added by: OpenID created token
URL: https://github.com/pypa/pip-audit/commit/d4242095300357730e1510ef2db837cf1b1142f1

File added to version 2.6.1
Filename: `pip_audit-2.6.1-py3-none-any.whl`
Added by: OpenID created token
URL: https://github.com/pypa/pip-audit/commit/d4242095300357730e1510ef2db837cf1b1142f1

Version 2.6.1 created
Added by: OpenID created token
URL: https://github.com/pypa/pip-audit/commit/d4242095300357730e1510ef2db837cf1b1142f1

sigstore

Takeaways

- **Trusted publishing is a double win: both for security *and* for usability**
  - The best kind of security improvements make users' lives easier, not harder!
  - Cynically: the *only* kinds of security improvements that matter are the ones that engineers want to use
- **Trusted publishing is not tied to PyPI; other package indices can use the same techniques and reap the same benefits!**
  - We (Trail of Bits) would be thrilled to reapply our experience on PyPI to other ecosystems; please come find me during the day and chat with me about it!
- **Trusted publishing is a logical step towards our shared supply chain goals: source and build provenance, code signing, generalized verifiable attestations over software/dependency graphs**

thank you!

- **these slides will soon be available here:**

  https://yossarian.net/publications#packagingcon-2023

- **resources:**
  - docs.pypi.org/trusted-publishers: official PyPI documentation for trusted publishing
  - “Introducing ‘Trusted Publishers’”: official PyPI announcement post
  - "Trusted publishing: a new benchmark for packaging security": ToB’s writeup on trusted publishing, threat modeling, etc.
- **Contact:**
  - william@trailofbits.com
  - @yossarian@infosec.exchange