SUNIMOD FIELD NOTES

Your Private Package Can Be Replaced: How Dependency Confusion Hijacks a Software Build.

A mixed public-and-private package configuration can let an untrusted release outrank an approved internal dependency. Learn how dependency confusion works, reproduce it safely on localhost, assess the business impact, and harden the path from source code to release.

BY Christopher Jacobson

PUBLISHED July 16, 2026

TOPIC CISSP, Domain 1

https://sunimod.com/your-private-package-can-be-replaced-how-dependency-confusion-hijacks-a-software-build/

The central lesson

A private package name is not a trust control. The real control is the complete path that tells a resolver which repositories it may search, how it chooses among matching candidates, which exact artifact is acceptable, and what privileges are available when package code runs.

This article uses Python and `pip` because the behavior is easy to demonstrate clearly. The larger lesson applies across software ecosystems: never assume that "private" and "public" sources remain separate unless the tooling and network design enforce that separation.

Educational guidance from Sunimod. The lab creates harmless packages, listens only on `127.0.0.1`, uses reserved example hosts, contacts no public package registry, and writes only a visible local marker. Test only systems and repositories you own or are explicitly authorized to assess.

What you will learn

1. What dependency confusion is

Dependency confusion occurs when a developer tool or build system searches more than one package source for a dependency, finds packages with the same normalized distribution name, and selects a candidate from an untrusted source instead of the organization's intended private source.

The weakness is not that the package manager "breaks into" the private repository. The private repository may be functioning correctly. The weakness is that the resolver has been told to treat candidates from different trust levels as one pool.

Consider an internal distribution called `examplecorp-private-rates-7f31c2`. The company publishes version `1.4.2` in its private repository. A second repository exposes version `99.0.0` under the same

normalized name. A requirement such as `>=1.4.2` accepts both versions. When the resolver compares both repositories and chooses the best matching version, the untrusted `99.0.0` candidate can win.

> **Dependency confusion is not typosquatting**
>
> Typosquatting relies on a similar-looking misspelling. Dependency confusion uses the same normalized distribution name the build already expects. A developer may make no spelling mistake at all.

The business problem behind the code problem

A build pipeline converts source code into something the business may deploy, sell, host, sign, or deliver to customers. Every dependency selected during that process becomes part of the business's product and operational risk. A resolver configuration is therefore not merely a developer preference; it is a control over which outside code may enter a trusted production path.

This makes the issue a practical, solvable service opportunity: inventory package sources, document the intended trust rules, reproduce actual resolver behavior in a controlled environment, remove mixed-source ambiguity, verify artifacts, reduce build privileges, and preserve evidence about how each release was assembled.

2. The resolver trust mismatch

People often read `--index-url` as "primary and trusted" and `--extra-index-url` as "fallback." That mental model is unsafe. The current pip documentation states that searched locations have no priority: candidates are collected from the configured index, extra indexes, local files, and `--find-links` locations, then the best matching version is selected.

The organization sees two categories—private code and public code. The resolver sees matching candidates. The vulnerability exists in the gap between those two models.

Candidate selection

For a requirement such as `examplecorp-private-rates-7f31c2>=1.4.2`, the simplified reasoning is:

1. **Normalize the requested distribution name.** Case and runs of periods, underscores, and hyphens are normalized for comparison.
2. **Search every configured location.** A private index, an additional index, and applicable file locations may all contribute candidates.
3. **Discard incompatible candidates.** Version constraints, Python compatibility, platform tags, and other rules reduce the set.
4. **Select the preferred candidate.** In the common case, the latest compatible version satisfying the requirement is selected.
5. **Obtain and install the distribution.** A source distribution may invoke build tooling; a wheel is unpacked and then commonly imported by tests or the application.

The source's business trust level is not automatically part of that comparison. Unless architecture and policy make source trust enforceable, a higher untrusted version can be technically valid.

Exact version pinning is not source pinning

Changing `>=1.4.2` to `==1.4.2` is valuable for repeatability, but it does not by itself identify which repository or artifact is trusted. pip documents that when more than one source offers the selected version, any source is assumed acceptable. If an untrusted source exposes the same version, a version-only pin still leaves source ambiguity.

Plaintext · an exact version with mixed sources is still ambiguous

```
--index-url https://packages.example.invalid/simple
--extra-index-url https://public-packages.example.invalid/simple

examplecorp-private-rates-7f31c2==1.4.2
```

A cryptographic hash binds the requirement to exact bytes. A controlled repository path prevents an untrusted source from participating in candidate selection. Durable protection uses both ideas instead of treating either one as a universal solution.

Distribution names and import names are different

The name used in `pip install` identifies a distribution. The name used in a Python `import` statement identifies an import package. Python packaging does not enforce a one-to-one naming relationship between them. An untrusted distribution can therefore provide the import name an application expects, preserve the expected functions, and add unwanted side effects.

Name normalization creates equivalent spellings

Python package indexes compare normalized distribution names. Case is lowered, and runs of `.`, `_`, or `-` become one hyphen. These spellings are equivalent for distribution lookup:

Python · show equivalent distribution names

```python
#!/usr/bin/env python3
from __future__ import annotations

import re


def normalize_project_name(value: str) -> str:
    return re.sub(r"[-_.]+", "-", value).lower()


candidates = (
    "ExampleCorp.Private_Rates",
    "examplecorp-private-rates",
    "EXAMPLECORP__PRIVATE...RATES",
    "examplecorp--private_rates",
)

for candidate in candidates:
    print(f"{candidate:34} -> {normalize_project_name(candidate)}")
```

Plaintext · normalization output

```
ExampleCorp.Private_Rates          -> examplecorp-private-rates
examplecorp-private-rates          -> examplecorp-private-rates
EXAMPLECORP__PRIVATE...RATES       -> examplecorp-private-rates
examplecorp--private_rates         -> examplecorp-private-rates
```

A review that compares raw strings but never normalizes them can miss collisions. Inventory tooling should store both the declared spelling and the normalized form.

3. The attack chain

Dependency confusion usually succeeds through a sequence of ordinary engineering events. The chain matters because each step offers a control point.

1. **An internal distribution name becomes knowable.** It may appear in a manifest, lockfile, source archive, container layer, build log, stack trace, support bundle, documentation, employee workstation, or third-party integration. Discovery does not necessarily require access to the private package repository.
2. **An untrusted repository accepts the same normalized name.** The release may use a conspicuously high stable version or, in a more targeted scenario, the same version the company pins.
3. **The build searches multiple trust levels.** Configuration may come from a requirements file, command line, environment variable, user-level `pip.conf`, runner image, container layer, or organization-wide automation.
4. **The resolver accepts the untrusted candidate.** A broad version range makes a higher version effective. A version-only pin may remain ambiguous when identical versions exist in multiple sources.
5. **The package reaches an execution point.** Source build hooks, metadata generation, tests, imports, command entry points, or application startup cause code from the selected distribution to run.
6. **The execution context determines the blast radius.** A developer account, persistent self-hosted runner, release signer, package publisher, cloud deployment identity, or production host may each expose different authority.
7. **The resulting artifact may move downstream.** If the build succeeds, the compromised component can be included in an image, wheel, ZIP archive, web deployment, desktop installer, or customer release.

> **A successful build can be the dangerous signal**
>
> A malicious replacement does not have to crash. It can preserve the functions the application expects, perform an unwanted action quietly, and let tests pass. Build success proves compatibility with the test suite; it does not prove provenance or trust.

A useful way to reason about severity is:

Potential impact = execution point × privileges × reachable secrets × network access × persistence × downstream distribution.

A temporary test environment with no credentials and blocked outbound access has a much smaller blast radius than a long-lived release runner that can write to source repositories, publish packages, sign artifacts, deploy production, and reach internal services.

4. A vulnerable build configuration

The following fictitious `requirements.txt` demonstrates the risky design. Both hosts use the reserved `.invalid` suffix and will not provide real packages. Do not replace them with live systems for testing; use the localhost lab in the next section.

Plaintext · intentionally vulnerable requirements.txt

```
--index-url https://packages.example.invalid/simple
--extra-index-url https://public-packages.example.invalid/simple

examplecorp-private-rates-7f31c2>=1.4.2
```

The developer may intend the first source to supply private packages and the second source to supply public packages. The resolver is not being given an enforceable rule that says, “This name may come only from the private source.” Both sources participate in selection.

Where the configuration can hide

- A repository requirements file with `--extra-index-url`.
- A CI environment variable such as `PIP_EXTRA_INDEX_URL`.
- A global or user `pip.conf` baked into a runner or container image.
- A shell alias, wrapper, Makefile, task runner, or deployment script.
- A `--find-links` wheel location used while the default public index remains enabled.
- A dependency-management service that proxies multiple upstream repositories without an enforceable allowlist or source rule.

Repository review alone is therefore necessary but not always sufficient. The effective pip configuration must be inspected in the same identity and runtime context used by the build.

> **Transitive and build dependencies count too**
>
> Top-level requirements are only the visible edge of the graph. Runtime dependencies can request more packages, and source builds can declare separate build-system dependencies. Controls must cover the full resolved graph and the tools used to create it.

5. Safe localhost demonstration on Ubuntu

This lab builds two valid but harmless Python wheels with the same distribution name. Version `1.4.2` is placed in a simulated private repository. Version `99.0.0` is placed in a simulated public repository. Both are served by temporary HTTP servers bound only to `127.0.0.1`.

The mixed-source environment selects `99.0.0`. A second environment that can see only the private repository selects `1.4.2`. The simulated public package writes a harmless text marker when imported so the result is unambiguous.

Lab safety properties

- No public registry is contacted.

- No package is uploaded anywhere.
- No credentials, environment secrets, browser data, or network files are read.
- Both package repositories listen only on the local loopback interface.
- The only side effect is a text marker under the current user's cache directory.
- The script refuses to overwrite an existing lab directory.

Install the Ubuntu prerequisite

Bash · install Python virtual-environment support on Ubuntu

```
sudo apt update
sudo apt install python3-venv
```

Create the lab script

Save the complete file as `build_dependency_confusion_lab.py`.

Python · build_dependency_confusion_lab.py

```
#!/usr/bin/env python3
"""Build and run a localhost-only dependency-confusion demonstration.

The lab creates two harmless wheel files with the same distribution name:
- private repository: version 1.4.2
- simulated public repository: version 99.0.0

Both repositories are served only on 127.0.0.1. The simulated public wheel
writes a text marker when imported so the selected source is visible. No
external package index is contacted and no credentials are used.
"""

from __future__ import annotations

import base64
import csv
import hashlib
import http.server
import io
import os
import subprocess
import sys
import textwrap
import threading
import venv
import zipfile
from functools import partial
from pathlib import Path
from typing import Iterable

DISTRIBUTION = "examplecorp-private-rates-7f31c2"
MODULE = "examplecorp_private_rates_7f31c2"
PRIVATE_VERSION = "1.4.2"
SIMULATED_PUBLIC_VERSION = "99.0.0"
LAB_ROOT = Path.home() / "sunimod-dependency-confusion-lab"


def wheel_name(version: str) -> str:
    return f"{MODULE}-{version}-py3-none-any.whl"


def record_hash(content: bytes) -> str:
    digest = hashlib.sha256(content).digest()
    encoded = base64.urlsafe_b64encode(digest).rstrip(b"=").decode("ascii")
```

```
    return f"sha256={encoded}"


def write_wheel(repository_root: Path, version: str, origin: str, marker: bool) -> Path:
    project_dir = repository_root / "simple" / DISTRIBUTION
    project_dir.mkdir(parents=True, exist_ok=True)

    dist_info = f"{MODULE}-{version}.dist-info"
    archive_path = project_dir / wheel_name(version)

    module_lines = [
        '"""Harmless package generated for the Sunimod localhost lab."""',
        "",
        "from pathlib import Path",
        "",
        f'__version__ = "{version}"',
        f'ORIGIN = "{origin}"',
        "",
    ]
    if marker:
        module_lines.extend(
            [
                'marker_dir = Path.home() / ".cache" / "sunimod-dependency-confusion-lab"',
                "marker_dir.mkdir(parents=True, exist_ok=True)",
                '(marker_dir / "selected-package.txt").write_text(',
                '    "The simulated public repository package was imported.\\n",',
                '    encoding="utf-8",',
                ')',
                "",
            ]
        )
    module_lines.extend(
        [
            "def describe() -> str:",
            '    return f"{ORIGIN} version {__version__}"',
            "",
        ]
    )
    module_source = "\n".join(module_lines).encode("utf-8")

    metadata = textwrap.dedent(
        f'''\
        Metadata-Version: 2.1
        Name: {DISTRIBUTION}
        Version: {version}
        Summary: Harmless localhost-only dependency-confusion lab package
        Requires-Python: >=3.10

        '''
    ).encode("utf-8")

    wheel_metadata = textwrap.dedent(
        '''\
        Wheel-Version: 1.0
        Generator: sunimod-localhost-lab
        Root-Is-Purelib: true
        Tag: py3-none-any

        '''
    ).encode("utf-8")

    top_level = f"{MODULE}\n".encode("utf-8")

    files: list[tuple[str, bytes]] = [
        (f"{MODULE}/__init__.py", module_source),
        (f"{dist_info}/METADATA", metadata),
        (f"{dist_info}/WHEEL", wheel_metadata),
        (f"{dist_info}/top_level.txt", top_level),
    ]

    record_buffer = io.StringIO(newline="")
    writer = csv.writer(record_buffer, lineterminator="\n")
```

```
    for name, content in files:
        writer.writerow((name, record_hash(content), str(len(content))))
    record_name = f"{dist_info}/RECORD"
    writer.writerow((record_name, "", ""))
    files.append((record_name, record_buffer.getvalue().encode("utf-8")))

    with zipfile.ZipFile(archive_path, "w", compression=zipfile.ZIP_DEFLATED) as archive:
        for name, content in files:
            info = zipfile.ZipInfo(name, date_time=(2020, 1, 1, 0, 0, 0))
            info.compress_type = zipfile.ZIP_DEFLATED
            info.external_attr = 0o644 << 16
            archive.writestr(info, content)

    return archive_path


class QuietHandler(http.server.SimpleHTTPRequestHandler):
    def log_message(self, format: str, *args: object) -> None:
        return


def start_repository(repository_root: Path) -> tuple[http.server.ThreadingHTTPServer,
    threading.Thread]:
    handler = partial(QuietHandler, directory=str(repository_root))
    server = http.server.ThreadingHTTPServer(("127.0.0.1", 0), handler)
    thread = threading.Thread(target=server.serve_forever, daemon=True)
    thread.start()
    return server, thread


def run(command: Iterable[str], *, env: dict[str, str] | None = None) ->
    subprocess.CompletedProcess[str]:
    completed = subprocess.run(
        list(command),
        check=True,
        text=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        env=env,
    )
    print(completed.stdout.rstrip())
    return completed


def create_environment(path: Path) -> Path:
    venv.EnvBuilder(with_pip=True, clear=True).create(path)
    return path / "bin" / "python"


def install_and_report(
    python: Path,
    private_url: str,
    simulated_public_url: str | None,
    requirement: str,
) -> None:
    command = [
        str(python),
        "-m",
        "pip",
        "install",
        "--disable-pip-version-check",
        "--no-cache-dir",
        "--no-deps",
        "--index-url",
        private_url,
    ]
    if simulated_public_url is not None:
        command.extend(("--extra-index-url", simulated_public_url))
    command.append(requirement)

    env = {
        key: value
```

```
        for key, value in os.environ.items()
        if not key.upper().startswith("PIP_")
    }
    env.pop("PYTHONHOME", None)
    env.pop("PYTHONPATH", None)
    env["PIP_CONFIG_FILE"] = os.devnull
    run(command, env=env)

    probe = (
        "from importlib.metadata import version; "
        f"import {MODULE}; "
        f"print('selected:', {MODULE}.describe()); "
        f"print('installed version:', version('{DISTRIBUTION}'))"
    )
    run((str(python), "-c", probe), env=env)


def main() -> int:
    if LAB_ROOT.exists():
        print(f"Refusing to overwrite existing lab directory: {LAB_ROOT}", file=sys.stderr)
        print("Remove that directory manually after reviewing it, then run the lab again.",
            file=sys.stderr)
        return 2

    private_repo = LAB_ROOT / "private-repository"
    simulated_public_repo = LAB_ROOT / "simulated-public-repository"
    mixed_venv = LAB_ROOT / "mixed-sources-venv"
    private_venv = LAB_ROOT / "private-only-venv"

    private_wheel = write_wheel(
        private_repo,
        PRIVATE_VERSION,
        "private-repository",
        marker=False,
    )
    simulated_public_wheel = write_wheel(
        simulated_public_repo,
        SIMULATED_PUBLIC_VERSION,
        "simulated-public-repository",
        marker=True,
    )

    private_server, private_thread = start_repository(private_repo)
    public_server, public_thread = start_repository(simulated_public_repo)

    private_url = f"http://127.0.0.1:{private_server.server_port}/simple"
    public_url = f"http://127.0.0.1:{public_server.server_port}/simple"

    try:
        print(f"Created private wheel: {private_wheel}")
        print(f"Created simulated public wheel: {simulated_public_wheel}")
        print(f"Private repository: {private_url}")
        print(f"Simulated public repository: {public_url}")

        print("\n=== Vulnerable mixed-source install ===")
        mixed_python = create_environment(mixed_venv)
        install_and_report(
            mixed_python,
            private_url,
            public_url,
            f"{DISTRIBUTION}>={PRIVATE_VERSION}",
        )

        print("\n=== Private-only control install ===")
        private_python = create_environment(private_venv)
        install_and_report(
            private_python,
            private_url,
            None,
            f"{DISTRIBUTION}=={PRIVATE_VERSION}",
        )
```

```
        marker = Path.home() / ".cache" / "sunimod-dependency-confusion-lab" /
            "selected-package.txt"
        print("\n=== Harmless marker check ===")
        if marker.exists():
            print(marker.read_text(encoding="utf-8").rstrip())
            print(f"Marker path: {marker}")
        else:
            print("The simulated public package marker was not created.")
            return 1

        print("\nThe lab used localhost only. No public registry was contacted.")
        print(f"Review the files under {LAB_ROOT}, then remove that directory when
            finished.")
        return 0
    finally:
        private_server.shutdown()
        public_server.shutdown()
        private_server.server_close()
        public_server.server_close()
        private_thread.join(timeout=5)
        public_thread.join(timeout=5)


if __name__ == "__main__":
    raise SystemExit(main())
```

Run the lab

Bash · execute the localhost-only demonstration

```
python3 build_dependency_confusion_lab.py
```

The port numbers are selected dynamically, so they will differ. A successful run resembles this tested output:

Plaintext · tested lab output

```
=== Vulnerable mixed-source install ===
Looking in indexes: http://127.0.0.1:40661/simple, http://127.0.0.1:35235/simple
Collecting examplecorp-private-rates-7f31c2>=1.4.2
  Downloading http://127.0.0.1:35235/simple/examplecorp-private-rates-7f31c2/examplecorp_pri
      vate_rates_7f31c2-99.0.0-py3-none-any.whl
Successfully installed examplecorp-private-rates-7f31c2-99.0.0
selected: simulated-public-repository version 99.0.0
installed version: 99.0.0

=== Private-only control install ===
Looking in indexes: http://127.0.0.1:40661/simple
Collecting examplecorp-private-rates-7f31c2==1.4.2
  Downloading http://127.0.0.1:40661/simple/examplecorp-private-rates-7f31c2/examplecorp_pri
      vate_rates_7f31c2-1.4.2-py3-none-any.whl
Successfully installed examplecorp-private-rates-7f31c2-1.4.2
selected: private-repository version 1.4.2
installed version: 1.4.2

The lab used localhost only. No public registry was contacted.
```

What the result proves

The private repository was online, contained the expected name, and offered a compatible package. Authentication failure was not the issue. The mixed-source resolver still selected the higher version from the other repository because both locations contributed candidates.

The control run changed one architectural fact: the untrusted repository was not a candidate source. The expected private package then won without relying on a convention about source order.

> **Turn the lab into a verified install**
>
> After the lab finishes, the following complete Ubuntu script computes a SHA-256 hash of the trusted local wheel, writes an exact lock entry, disables indexes, allows only wheels, and requires the local hash. Save it as `install_verified_lab_package.sh` and run it with `bash install_verified_lab_package.sh`.

Bash · install only the verified private lab artifact

```bash
#!/usr/bin/env bash
set -Eeuo pipefail

LAB_ROOT="${HOME}/sunimod-dependency-confusion-lab"
PACKAGE="examplecorp-private-rates-7f31c2"
VERSION="1.4.2"
WHEEL_DIR="${LAB_ROOT}/private-repository/simple/${PACKAGE}"
WHEEL="${WHEEL_DIR}/examplecorp_private_rates_7f31c2-${VERSION}-py3-none-any.whl"
LOCKFILE="${LAB_ROOT}/requirements.lock"
VENV="${LAB_ROOT}/verified-venv"

if [[ ! -f "${WHEEL}" ]]; then
  printf 'Trusted wheel not found: %s\n' "${WHEEL}" >&2
  exit 1
fi

HASH="$(sha256sum "${WHEEL}" | awk '{print $1}')"
printf '%s==%s --hash=sha256:%s\n' \
  "${PACKAGE}" "${VERSION}" "${HASH}" > "${LOCKFILE}"

python3 -m venv "${VENV}"
"${VENV}/bin/python" -m pip install \
  --disable-pip-version-check \
  --no-index \
  --find-links "file://${WHEEL_DIR}" \
  --only-binary :all: \
  --require-hashes \
  -r "${LOCKFILE}"

"${VENV}/bin/python" -c \
  'import examplecorp_private_rates_7f31c2 as package; print(package.describe())'
```

This small lab has one dependency and one platform-independent wheel. A production lock process must enumerate and hash every permitted direct, transitive, and build dependency, including all platform artifacts the deployment actually supports. Hashes should be generated in a controlled resolution process and reviewed as security-sensitive changes.

6. How package code reaches execution

“The package was only downloaded” is not a reliable boundary. Package workflows contain several execution points, and the exact point depends on the distribution format and build process.

Source distribution builds

A source distribution may require pip to create a build environment, obtain build-system dependencies, invoke a build backend, and generate metadata or a wheel. That process executes code involved in the build. This is one reason pip's secure-install guidance recommends disallowing source distributions with `--only-binary :all:` where operationally feasible.

Wheel installation and import

A pure Python wheel is generally unpacked rather than built from source, but its modules become executable application code. Test discovery, health checks, migration jobs, command entry points, or normal application startup may import it immediately. The localhost lab uses this path: installation succeeds, and the probe import executes the harmless marker action.

Compatibility can conceal the replacement

An untrusted distribution can expose the same import package, functions, and return values expected by the application. The unwanted behavior can occur before or after the legitimate-looking operation. That is why unit tests focused only on functional output may pass.

Why wheels-only is not a complete fix

`--only-binary :all:` removes source-build execution from the install path, which is useful risk reduction. It does not make an untrusted wheel safe, identify its source, or prevent its modules from running after installation. Source restriction, exact artifacts, hashes, controlled build privileges, and post-build evidence remain necessary.

7. How to find exposure without executing packages

Begin with a read-only configuration review. Do not test by publishing a lookalike package to a public service, and do not install unknown candidates merely to see what happens. The initial objective is to learn what sources the build can reach and how those sources are configured.

Manual review checklist

- Search requirements files, build scripts, Dockerfiles, CI definitions, task runners, and environment templates for `--extra-index-url`, `PIP_EXTRA_INDEX_URL`, `--find-links`, and `PIP_FIND_LINKS`.
- Inspect system, user, virtual-environment, container, and runner-image pip configuration.
- Run `python3 -m pip config debug` in the authorized build context to see which configuration files contribute settings.
- Record every private distribution's declared spelling, normalized name, owner, repository, approved versions, and expected artifact hashes.
- Review direct, transitive, optional, development, test, and build-system dependencies.
- Determine whether build workers can reach public package services directly.
- Review the permissions and secrets available during dependency resolution, build, test, signing, publishing, and deployment.

- Preserve pip reports, package-manager logs, lockfiles, SBOMs, artifact digests, and provenance evidence for releases.

A read-only repository audit tool

The following complete Python program scans text configuration in an authorized repository. It flags additional indexes, unsafe `--find-links` combinations, trusted-host overrides, unpinned internal requirements, missing local SHA-256 constraints, and normalized-name collisions.

It does not contact any repository, download any artifact, import any dependency, or execute package code. It is deliberately narrow: a clean result means only that these patterns were not found in the scanned files.

Python · audit_pip_sources.py

```
#!/usr/bin/env python3
"""Read-only audit for pip dependency-source trust risks.

The tool inspects repository configuration and requirements files. It never
contacts a package repository, downloads a distribution, or executes package
code. Findings are configuration-review prompts rather than proof of compromise.
"""

from __future__ import annotations

import argparse
import re
import sys
from dataclasses import dataclass
from pathlib import Path
from typing import Iterable

SKIP_DIRECTORIES = {
    ".git",
    ".hg",
    ".svn",
    ".tox",
    ".venv",
    "venv",
    "node_modules",
    "vendor",
    "dist",
    "build",
    "__pycache__",
}

SCANNED_FILENAMES = {
    "pip.conf",
    "pip.ini",
    "pyproject.toml",
    "setup.cfg",
    "tox.ini",
    "Dockerfile",
}

SCANNED_SUFFIXES = {
    ".txt",
    ".in",
    ".cfg",
    ".ini",
    ".toml",
    ".yaml",
    ".yml",
    ".sh",
    ".bash",
    ".env",
}
```

```
REQUIREMENTS_NAME = re.compile(r"^requirements(?:[-_.].*)?\.(?:txt|in)$", re.IGNORECASE)
PROJECT_NAME = re.compile(r"^([A-Za-z0-9](?:[A-Za-z0-9._-]*[A-Za-z0-9])?)")
EXTRA_INDEX_PATTERNS = (
    re.compile(r"(?:^|\s)--extra-index-url(?:\s|=)", re.IGNORECASE),
    re.compile(r"\bPIP_EXTRA_INDEX_URL\b", re.IGNORECASE),
    re.compile(r"^\s*extra-index-url\s*=", re.IGNORECASE),
)
FIND_LINKS_PATTERNS = (
    re.compile(r"(?:^|\s)--find-links(?:\s|=)", re.IGNORECASE),
    re.compile(r"\bPIP_FIND_LINKS\b", re.IGNORECASE),
    re.compile(r"^\s*find-links\s*=", re.IGNORECASE),
)
NO_INDEX_PATTERNS = (
    re.compile(r"(?:^|\s)--no-index(?:\s|$)", re.IGNORECASE),
    re.compile(r"\bPIP_NO_INDEX\s*=\s*(?:1|true|yes|on)\b", re.IGNORECASE),
    re.compile(r"^\s*no-index\s*=\s*(?:1|true|yes|on)\s*$", re.IGNORECASE),
)
TRUSTED_HOST_PATTERNS = (
    re.compile(r"(?:^|\s)--trusted-host(?:\s|=)", re.IGNORECASE),
    re.compile(r"\bPIP_TRUSTED_HOST\b", re.IGNORECASE),
    re.compile(r"^\s*trusted-host\s*=", re.IGNORECASE),
)


@dataclass(frozen=True)
class Finding:
    severity: str
    path: Path
    line: int
    message: str


def normalize_project_name(value: str) -> str:
    return re.sub(r"[-_.]+", "-", value).lower()


def iter_files(root: Path) -> Iterable[Path]:
    for path in root.rglob("*"):
        if not path.is_file():
            continue
        if any(part in SKIP_DIRECTORIES for part in path.relative_to(root).parts):
            continue
        if (
            path.name in SCANNED_FILENAMES
            or path.name.startswith("Dockerfile.")
            or path.suffix.lower() in SCANNED_SUFFIXES
        ):
            yield path


def logical_lines(text: str) -> Iterable[tuple[int, str]]:
    start_line = 1
    buffer: list[str] = []
    for line_number, raw_line in enumerate(text.splitlines(), start=1):
        stripped = raw_line.rstrip()
        if not buffer:
            start_line = line_number
        if stripped.endswith("\\"):
            buffer.append(stripped[:-1].strip())
            continue
        buffer.append(stripped.strip())
        yield start_line, " ".join(part for part in buffer if part)
        buffer = []
    if buffer:
        yield start_line, " ".join(part for part in buffer if part)


def is_internal(name: str, prefixes: tuple[str, ...]) -> bool:
    normalized = normalize_project_name(name)
    return any(
        normalized == prefix or normalized.startswith(prefix + "-")
```

```
        for prefix in prefixes
    )


def inspect_requirements(
    path: Path,
    text: str,
    prefixes: tuple[str, ...],
) -> list[Finding]:
    findings: list[Finding] = []
    seen_spellings: dict[str, tuple[str, int]] = {}

    for line_number, logical in logical_lines(text):
        content = logical.strip()
        if not content or content.startswith(("#", ";", "-", "git+", "http://",
            "https://")):
            continue

        match = PROJECT_NAME.match(content)
        if not match:
            continue

        raw_name = match.group(1)
        normalized = normalize_project_name(raw_name)
        previous = seen_spellings.get(normalized)
        if previous and previous[0] != raw_name:
            findings.append(
                Finding(
                    "MEDIUM",
                    path,
                    line_number,
                    (
                        f"{raw_name!r} normalizes to the same project name as "
                        f"{previous[0]!r} on line {previous[1]}."
                    ),
                )
            )
        else:
            seen_spellings[normalized] = (raw_name, line_number)

        if not is_internal(raw_name, prefixes):
            continue

        exact_pin = re.search(r"(?<![<>=!~])==(?!=)\s*[^\s;,]+", content) is not None
        arbitrary_exact = re.search(r"===\s*[^\s;,]+", content) is not None
        direct_reference = re.search(r"\s@\s\S+", content) is not None
        sha256_hash = "--hash=sha256:" in content.lower()

        if not (exact_pin or arbitrary_exact or direct_reference):
            findings.append(
                Finding(
                    "MEDIUM",
                    path,
                    line_number,
                    f"Internal dependency {raw_name!r} is not pinned to an exact version or
                        direct artifact.",
                )
            )

        if not sha256_hash:
            findings.append(
                Finding(
                    "MEDIUM",
                    path,
                    line_number,
                    f"Internal dependency {raw_name!r} has no local SHA-256 hash
                        constraint.",
                )
            )

    return findings
```

```
def inspect_file(path: Path, prefixes: tuple[str, ...]) -> list[Finding]:
    try:
        text = path.read_text(encoding="utf-8", errors="replace")
    except OSError as exc:
        return [Finding("INFO", path, 0, f"Could not read file: {exc}")]

    findings: list[Finding] = []
    file_disables_indexes = any(pattern.search(text) for pattern in NO_INDEX_PATTERNS)

    for line_number, logical in logical_lines(text):
        content = logical.strip()
        if not content or content.startswith(("#", ";")):
            continue

        if any(pattern.search(content) for pattern in EXTRA_INDEX_PATTERNS):
            findings.append(
                Finding(
                    "HIGH",
                    path,
                    line_number,
                    "Additional package index configured; candidates may be selected across
                        trust boundaries.",
                )
            )

        uses_find_links = any(pattern.search(content) for pattern in FIND_LINKS_PATTERNS)
        line_disables_indexes = any(pattern.search(content) for pattern in
            NO_INDEX_PATTERNS)
        if uses_find_links and not (file_disables_indexes or line_disables_indexes):
            findings.append(
                Finding(
                    "HIGH",
                    path,
                    line_number,
                    "Find-links source is configured without an evident no-index control;
                        pip may compare it with an index.",
                )
            )

        if any(pattern.search(content) for pattern in TRUSTED_HOST_PATTERNS):
            findings.append(
                Finding(
                    "MEDIUM",
                    path,
                    line_number,
                    "Trusted-host override found; verify that transport verification is not
                        being weakened.",
                )
            )

    if REQUIREMENTS_NAME.match(path.name):
        findings.extend(inspect_requirements(path, text, prefixes))

    return findings


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description=(
            "Read-only audit for pip source-mixing and internal dependency pinning risks. "
            "The tool never contacts a package repository or executes package code."
        )
    )
    parser.add_argument(
        "root",
        nargs="?",
        default=".",
        help="Repository directory to inspect (default: current directory).",
    )
    parser.add_argument(
        "--internal-prefix",
```

```
        action="append",
        required=True,
        help=(
            "Normalized prefix used by private distributions; repeat for multiple prefixes "
            "(example: --internal-prefix examplecorp)."
        ),
    )
    return parser.parse_args()


def main() -> int:
    args = parse_args()
    root = Path(args.root).resolve()
    if not root.is_dir():
        print(f"error: not a directory: {root}", file=sys.stderr)
        return 2

    prefixes = tuple(
        sorted({normalize_project_name(value) for value in args.internal_prefix})
    )
    findings: list[Finding] = []
    scanned = 0

    for path in iter_files(root):
        scanned += 1
        findings.extend(inspect_file(path, prefixes))

    severity_order = {"HIGH": 0, "MEDIUM": 1, "INFO": 2}
    findings.sort(
        key=lambda item: (
            severity_order.get(item.severity, 9),
            str(item.path),
            item.line,
            item.message,
        )
    )

    print("pip dependency-source audit")
    print("===========================")
    print(f"Root: {root}")
    print(f"Internal prefixes: {', '.join(prefixes)}")
    print(f"Files scanned: {scanned}")
    print()

    if not findings:
        print("No configured source-mixing or internal pinning findings were detected.")
        print("This is a focused configuration check, not proof that the supply chain is
            safe.")
        return 0

    for finding in findings:
        location = str(finding.path.relative_to(root))
        if finding.line:
            location = f"{location}:{finding.line}"
        print(f"[{finding.severity}] {location} - {finding.message}")

    if any(item.severity == "HIGH" for item in findings):
        return 2
    if any(item.severity == "MEDIUM" for item in findings):
        return 1
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

Run the audit tool

Use the prefixes your organization actually reserves for private distributions. Repeat the option for each prefix.

Bash · scan an authorized repository

```
python3 audit_pip_sources.py . \
  --internal-prefix examplecorp \
  --internal-prefix sunimod-internal
```

The process exits with status `2` when a high-severity source-mixing pattern is found, `1` for medium findings, and `0` when no configured pattern is detected. Example output:

Plaintext · audit findings

```
pip dependency-source audit
===========================
Root: /srv/examplecorp/application
Internal prefixes: examplecorp, sunimod-internal
Files scanned: 14

[HIGH] requirements.txt:2 - Additional package index configured; candidates may be selected
    across trust boundaries.
[MEDIUM] requirements.txt:4 - Internal dependency 'examplecorp_private_rates' has no local
    SHA-256 hash constraint.
[MEDIUM] requirements.txt:4 - Internal dependency 'examplecorp_private_rates' is not pinned
    to an exact version or direct artifact.
[MEDIUM] requirements.txt:5 - 'ExampleCorp.Private.Rates' normalizes to the same project
    name as 'examplecorp_private_rates' on line 4.
```

What the audit tool cannot prove

- It cannot see a runner's global configuration unless that configuration is included in the scan target.
- It cannot prove that a private package name is unclaimed or controlled on every external repository.
- It does not resolve dependency graphs or inspect build-system dependencies hidden behind generated metadata.
- It does not validate the provenance or contents of a wheel.
- It cannot determine whether a hash was generated from a genuinely trusted artifact.
- It does not replace review of network egress, package proxies, CI privileges, release evidence, or incident logs.

Use it as a repeatable first-pass control and a pull-request check, then validate the effective build environment.

8. Potential repercussions for your business or infrastructure

The business effect depends on where the package runs and what that identity can reach. The same replacement may be a contained test incident in one environment and a company-wide release incident in another.

> **Do not estimate impact from the package name alone.** Scope the exact version, artifact hash, source URL, installation time, execution point, runner identity, reachable network segments, available credentials, generated artifacts, and downstream releases.

Potential repercussion	What may happen	Business or infrastructure effect
Developer workstation compromise	Package code reads accessible project files, modifies local source, adds persistence, or uses the developer's authenticated tools.	Source integrity uncertainty, stolen work product, account misuse, and a wider investigation across repositories and endpoints.
CI runner compromise	The package accesses workspace files, caches, service containers, metadata endpoints, or credentials made available to the job.	Compromised builds, contaminated caches, repeated reinfection, and unauthorized access to connected systems.
Repository or workflow tampering	A write-capable automation token is used to alter code, tags, releases, workflows, or branch settings.	Persistent unauthorized changes, misleading history, emergency access review, and loss of confidence in prior releases.
Package or artifact publishing abuse	Publishing authority is used to replace internal packages or distribute a modified release.	Downstream compromise of teams, customers, or partners that trust the organization's release channel.
Production deployment	The selected package is embedded in a container, web application, desktop installer, scheduled job, or server deployment.	Unauthorized code reaches production, potentially expanding the incident to customer data and critical operations.
Credential and token exposure	Build or runtime credentials available to the process are copied or used.	Access to source control, cloud services, package repositories, monitoring, databases, or third-party services, limited by the credentials' permissions.
Data or intellectual-property loss	Source code, configuration, customer records, pricing logic, models, documents, or other accessible information is disclosed.	Privacy review, contract issues, competitive harm, customer impact, and difficult-to-measure long-term loss.
Service interruption	Builds fail, resources are modified, quotas are consumed, releases are halted, or affected systems are isolated for response.	Delayed launches, lost transactions, staff downtime, support volume, and recovery work.
Release integrity uncertainty	The organization cannot prove which dependency bytes entered each historical artifact.	Broader rebuild and redeployment scope, customer questions, and slower containment because every release requires validation.
Notification and contractual review	The event may involve regulated data, customer commitments, cyber-insurance terms, or vendor obligations.	Legal, privacy, insurance, leadership, customer, and partner coordination based on the facts and applicable obligations.
Incident-response expense	Teams preserve evidence, rotate credentials, rebuild runners, reissue artifacts, review logs, contact vendors, and validate production.	Unplanned labor and consulting cost, delayed roadmaps, management distraction, and lost trust.

These are potential outcomes, not a claim that every dependency-confusion event causes all of them. Impact should be established from evidence rather than assumed. Legal and notification duties depend on the affected data, contracts, jurisdictions, sector, insurance policy, and other case-specific facts.

9. What to do after suspected exposure

Treat a suspicious dependency selection as an integrity incident until evidence shows otherwise. Avoid deleting the runner, package cache, lockfile, or logs before preserving what investigators need.

1. **Stop promotion and deployment.** Pause affected build and release paths without destroying evidence. Block the suspect package name, version, hash, and source at controlled repositories and egress points.
2. **Preserve the exact evidence.** Save manifests, lockfiles, pip configuration, `pip --report` output, package-manager logs, wheel or source archives, hashes, SBOMs, provenance attestations, runner

image identifiers, job logs, and resulting artifacts.

3. **Identify the selected artifact.** Record its distribution name, normalized name, version, filename, cryptographic digest, source URL, install time, and every environment where it was obtained.
4. **Determine whether and when code executed.** Review source-build steps, test imports, entry points, application startup, migrations, post-build validation, and runtime deployment.
5. **Scope the execution context.** Determine the operating-system identity, repository permissions, cloud roles, package-publishing rights, signing access, deployment authority, network reach, mounted volumes, caches, and secrets available at that time.
6. **Contain credentials and persistence.** Preserve relevant logs, then revoke or rotate credentials the process could access. Inspect new users, keys, tokens, webhooks, scheduled jobs, workflow changes, package releases, cloud resources, and recovery-setting changes.
7. **Trace downstream artifacts.** Find every image, archive, package, deployment, or customer release built from the affected run or from a contaminated cache.
8. **Rebuild from a known-good boundary.** Use clean ephemeral infrastructure, one controlled package source, verified lock artifacts, fresh credentials, and known-good source revisions. Compare digests and behavior.
9. **Assess business obligations.** Involve security, engineering, leadership, privacy, legal, insurance, customers, and vendors as the evidence requires.
10. **Correct the trust design.** Remove mixed-source ambiguity, reduce privileges, improve evidence retention, and create a tested response runbook before restoring normal release velocity.

> **Do not fix the file and declare victory**
>
> Deleting `--extra-index-url` from one branch does not tell you whether the package executed, whether a runner-level setting remains, whether credentials were used, whether caches are contaminated, or whether an affected artifact was already deployed.

10. Designing a durable fix

No single flag solves every package-supply risk. The strongest practical design combines source control, artifact identity, privilege separation, build isolation, inventory, and evidence.

Use one controlled package entry point

Configure builds to use a single organization-controlled package repository or proxy as their only package entry point. That service should curate approved public packages, host private packages, enforce source and naming policy, retain access logs, and prevent direct public fallback.

- Disable direct internet package retrieval from build workers where feasible.
- Allow approved public dependencies through a controlled mirror or proxy.
- Make internal names available only from the intended private repository path.
- Define how new packages and versions are requested, reviewed, approved, and retired.
- Monitor and alert on denied or unexpected package requests.

A proxy is not automatically safe. A proxy that merges upstreams without enforceable package rules can reproduce the same confusion behind a different URL.

Reserve and inventory private names

Use a consistent organization-specific naming convention where the ecosystem supports it. Maintain an owner and lifecycle for every private distribution. Where authorized and appropriate, reserve corresponding names on relevant public services as a defense-in-depth measure.

Name reservation is not the primary boundary. It can fail through missed names, ownership lapse, new repositories, normalized variants, transitive dependencies, or future packages created before the reservation process runs.

Pin and hash every permitted artifact

Use exact versions or direct artifact references and enforce local SHA-256 hashes with `--require-hashes`. pip's hash-checking mode requires every dependency to be explicit, pinned, and hashed, which helps convert an open-ended resolution into an approved artifact set.

- Generate lock data in a controlled resolver environment.
- Review lockfile and hash changes as security-sensitive code changes.
- Include direct, transitive, optional, test, and build dependencies that enter the release process.
- Include hashes for every platform artifact the organization genuinely permits.
- Fail closed when a package, version, or hash is not approved.

A hash proves that downloaded bytes match the locally approved digest. It does not prove that the bytes were trustworthy when the digest was created. The approval process matters.

Prefer reviewed wheels where practical

Using `--only-binary :all:` avoids source distribution builds and their build-time execution path. Some dependencies or platforms may not publish acceptable wheels, so adoption requires inventory and testing. When source builds are necessary, perform them in a dedicated, restricted builder and promote the resulting reviewed wheel into the controlled repository.

Separate resolution, build, signing, and deployment

A dependency does not need deployment authority merely because it is required to compile or test software. Split the pipeline into distinct trust stages:

1. **Resolution stage:** obtains only approved artifacts, has no deployment credentials, and emits a reviewed lock or artifact set.
2. **Build stage:** consumes approved artifacts in an ephemeral environment with minimal network access and produces immutable output plus digests.
3. **Verification stage:** runs tests and policy checks against the exact output, not a newly resolved dependency set.
4. **Signing or attestation stage:** identifies the source revision, builder, inputs, and output digest without exposing signing material to user-defined build steps.

5. **Promotion and deployment stage:** receives only verified immutable artifacts and uses narrowly scoped, preferably short-lived deployment authority.

For GitHub Actions, set the default `GITHUB_TOKEN` permission to read-only and grant additional permissions only to the job that needs them. Keep dependency review and untrusted build steps away from package publishing and deployment secrets.

Use ephemeral, isolated builders

Destroy the build environment after each run. Do not let one build alter the next through a persistent workspace, user profile, globally installed package, or writable shared cache. Treat self-hosted runners as security-sensitive infrastructure with explicit patching, isolation, network, and reset controls.

Record what was built and how

Retain an SBOM or equivalent dependency inventory, source revision, lockfile, package-source evidence, artifact digests, builder identity, and provenance or attestation for each release. This evidence shortens incident scoping: the team can answer which artifact used which dependency bytes instead of rebuilding history from incomplete logs.

Monitor the control plane

- Alert on new package sources, source-order changes, trusted-host overrides, or disabled verification.
- Review new internal names for normalized collisions.
- Monitor package repository requests for unexpected names, versions, clients, or external fallbacks.
- Detect changes to runner images, global pip configuration, egress rules, and package proxy policy.
- Review package-publishing, signing, release, and deployment permissions regularly.
- Test incident response with a harmless local simulation before a real event forces the process.

11. A solvable Sunimod service

Sunimod can turn this risk into a defined improvement project: a **Software Supply-Chain and Build Trust Review**. The goal is not to hand over a scanner report. The goal is to make package selection understandable, enforceable, repeatable, and supportable in the environment your business actually uses.

Review scope

- Inventory manifests, lockfiles, private distributions, build-system requirements, package repositories, proxy settings, runner images, Dockerfiles, CI jobs, release scripts, and deployment handoffs.
- Map every effective package source, including repository files, environment variables, user and system configuration, container layers, and organization automation.
- Normalize private distribution names and identify naming collisions, inconsistent spellings, unclear ownership, and missing lifecycle records.
- Reproduce resolver behavior safely with local or controlled test repositories—never by publishing an unauthorized public package.

- Trace the privileges, credentials, network access, caches, and downstream artifacts associated with dependency resolution and build execution.

Implementation deliverables

- A documented package-source trust model and prioritized finding register.
- A single controlled repository or proxy design with public-package curation and private-package rules.
- Exact lock and hash enforcement appropriate to supported platforms.
- CI privilege reduction, stage separation, network restrictions, and ephemeral-build recommendations.
- Repository checks for source mixing, normalization collisions, and unreviewed lock changes.
- SBOM, digest, and provenance requirements matched to the release process.
- An incident-response runbook covering evidence preservation, credential containment, cache handling, release tracing, rebuild, and communication.
- Plain-language documentation and team walkthroughs so the controls remain usable after the project ends.

How success is measured

A useful outcome can be demonstrated:

- A build cannot retrieve packages directly from an unapproved external source.
- An internal distribution cannot be silently replaced by a higher version from another repository.
- An artifact with an unapproved digest fails closed.
- Dependency-resolution jobs lack publishing and deployment authority.
- Each release can be traced to a source revision, dependency set, builder, and output digest.
- The team can contain and investigate a simulated package-selection incident using its written runbook.

12. Frequently asked questions

Is a private repository enough?

No. A private repository protects its own contents and access. It does not prevent a resolver from also considering candidates from another configured source. The build must enforce where each approved package may come from.

Does an exact version stop the attack?

It prevents a higher version from satisfying the requirement, which is useful. It does not identify a trusted source or artifact. When the same selected version exists in more than one source, pip assumes either source is acceptable. Pair exact versions with source restriction and locally approved hashes.

Do hashes solve everything?

Hashes strongly bind a requirement to approved bytes and should be part of the design. They do not prove the artifact was safe when approved, protect a weak lock-generation process, reduce runner privileges, or provide release provenance by themselves.

Does wheels-only stop malicious code?

No. It removes source-build execution and reduces variability, but an untrusted wheel can still provide malicious importable code. Use it with source control, hashes, review, isolation, and least privilege.

Does reserving public names solve the problem?

It reduces one opportunity and can be worthwhile when permitted by service rules. It is defense in depth, not the main control. New internal names, expired ownership, other repositories, normalized variants, and transitive dependencies can escape the reservation process.

Does this only affect Python?

No. Package ecosystems differ in namespaces, resolver behavior, scopes, lock formats, scripts, signatures, and repository controls, but the general failure appears whenever software combines trusted and untrusted dependency sources without enforceable selection rules.

Can a clean scan prove the build is safe?

No. Static review may miss global runner configuration, generated manifests, proxy behavior, dynamic dependencies, compromised approved artifacts, or configuration supplied at runtime. Combine scanning with effective-configuration review, network policy, controlled test resolution, artifact verification, privilege analysis, and release evidence.

Should we test by publishing our private name publicly?

Not without explicit authorization, ownership review, service-rule review, and a controlled plan. Public publication can create legal, operational, and third-party risk. The localhost lab demonstrates the resolver behavior without claiming a public name or exposing anyone else.

13. Key takeaways

- A private package name is an identifier, not proof of trust.
- `--extra-index-url` is not a safe private-package fallback design in pip.
- pip compares candidates across configured locations rather than treating the first location as inherently trusted.
- Exact version pins improve repeatability but do not select a source or exact bytes.
- Locally approved SHA-256 hashes bind requirements to artifacts; the approval process must also be trustworthy.
- Wheels-only installation reduces source-build execution but does not make an untrusted wheel safe.

- Build privileges decide whether the incident stays local or reaches repositories, cloud services, production, and customers.
- One controlled package entry point, curated upstreams, ephemeral builders, stage separation, least privilege, SBOMs, and provenance create a stronger system together.
- A safe localhost simulation can validate the control without publishing or executing an unknown public package.

Sources and further reading

- pip documentation: pip install — candidate collection, version selection, index options, and the explicit warning about dependency confusion.
- pip documentation: Secure installs — hash-checking mode, exact requirements, dependency hashes, and wheels-only installation.
- Python Packaging User Guide: Names and normalization — the canonical normalization rule used for distribution names.
- Python Packaging User Guide: Distribution package versus import package — why installation names and import names are not an enforced one-to-one mapping.
- OWASP: Dependency Chain Abuse — attack paths, impact, controlled proxies, checksums, version locking, and build-context isolation.
- OWASP Software Supply Chain Security Cheat Sheet — build hardening, isolated ephemeral builds, provenance, final-artifact review, and inventory.
- NIST SP 800-218: Secure Software Development Framework — integrating secure development practices into the software lifecycle and addressing root causes.
- NIST SP 800-161 Rev. 1 Update 1 — identifying, assessing, and mitigating cybersecurity supply-chain risk.
- SLSA v1.2 Build Requirements — build provenance, hosted and isolated builds, and protection of signing material from user-defined build steps.
- GitHub Actions: Secure use reference — minimum token permissions, secret handling, workflow separation, and third-party action risk.

Product behavior and guidance can change. Validate controls against the versions, package services, build platforms, and legal obligations that apply to your environment.

Secure the path from source code to release

Hire Sunimod to review your dependency sources, private package names, lock and hash process, CI privileges, build isolation, release evidence, and incident-readiness. The result is a practical remediation plan—and, where appropriate, implementation work that makes the safer path the normal path.

Request a software supply-chain review

Describe the affected repositories, package ecosystems, build platform, and business outcome. Do not submit passwords, API keys, access tokens, signing keys, payment-card data, or other secrets through the form; Sunimod can arrange a safer handoff when access is required.