Protecting your organization against attacks via the build system

Louis Jacomet - Gradle Inc.

@ljacomet

Gradle

Louis Jacomet

Gradle, Inc.

Lead Software Engineer

Dependency Management / JVM Team at Gradle

Supply chain attacks are no longer an hypothesis

A supply-chain attack is an indirect attack which **targets the tools**, automatic software updates or supply chain in general, in order to introduce malicious code or dependencies into existing software, **without the developers being aware**.

The consequence of those attacks may be catastrophic as they are easily unnoticed and usually **scale out** because of the end targets: mobile applications for example.

There’s evidence of such attacks in the wild. Some are **suspected to be issued from Nation State Actors**.

CCleaner/Asus

- Hacked into (auto) update systems
- Tool update installed a backdoor (ShadowPad) on machines
- Performed a follow-up attack against high value targets

https://www.wired.com/story/inside-the-unnerving-supply-chain-attack-that-corrupted-ccleaner/

MS Tools attack

- Initial pattern similar to Asus / CCleaner, pointing to same attackers
- Microsoft developer tools were maliciously modified
- Game companies used the compromised dev tools to sign games

- Properly signed games are released with an embedded backdoor

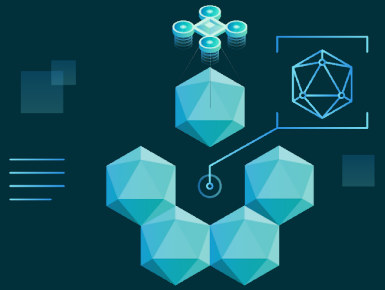
https://www.wired.com/story/supply-chain-hackers-videogames-asus-ccleaner/#

Homebrew - 30 minutes from nobody to a commit

- Case study by Eric Holmes in 2018
- Found a leaked GitHub key on CI
- Committed to the main repo

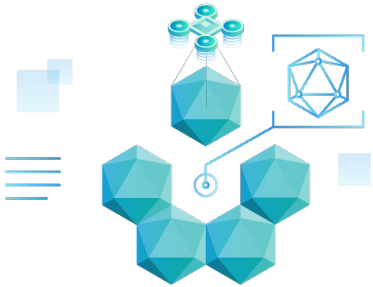
- ~~Profit~~ Responsible disclosure and get the project protected

https://medium.com/@vesirin/how-i-gained-commit-access-to-homebrew-in-30-minutes-2ae314df03ab



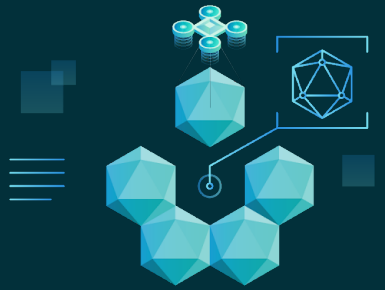
23% of all security advisories against JS projects refer to **intentionally malicious packages**

Jonathan Leitschuh, analyzing Npmjs advisories



“Maintainers of the RubyGems package repository have yanked **18 malicious versions of 11 Ruby libraries** that contained a backdoor mechanism and were caught inserting **code that launched hidden cryptocurrency mining operations** inside other people's Ruby projects.

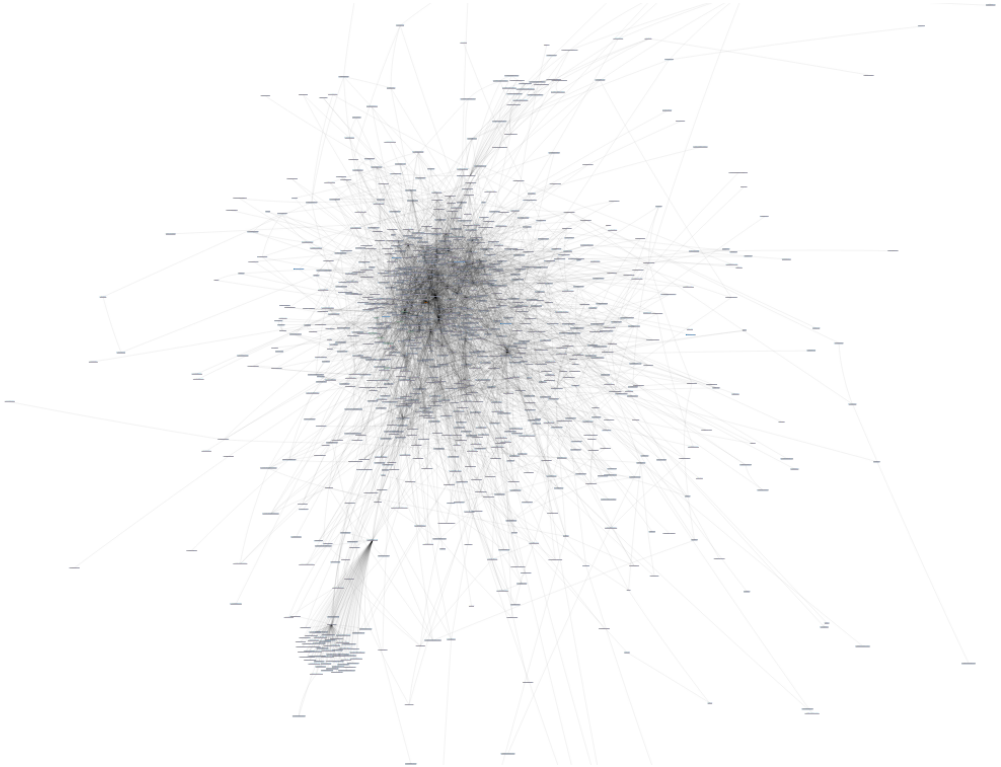
https://www.zdnet.com/article/backdoor-code-found-in-11-ruby-libraries/



“When installing an average npm package, a user implicitly **trusts around 80 other packages** due to transitive dependencies.

https://blog.acolyer.org/2019/09/30/small-world-with-high-risks/

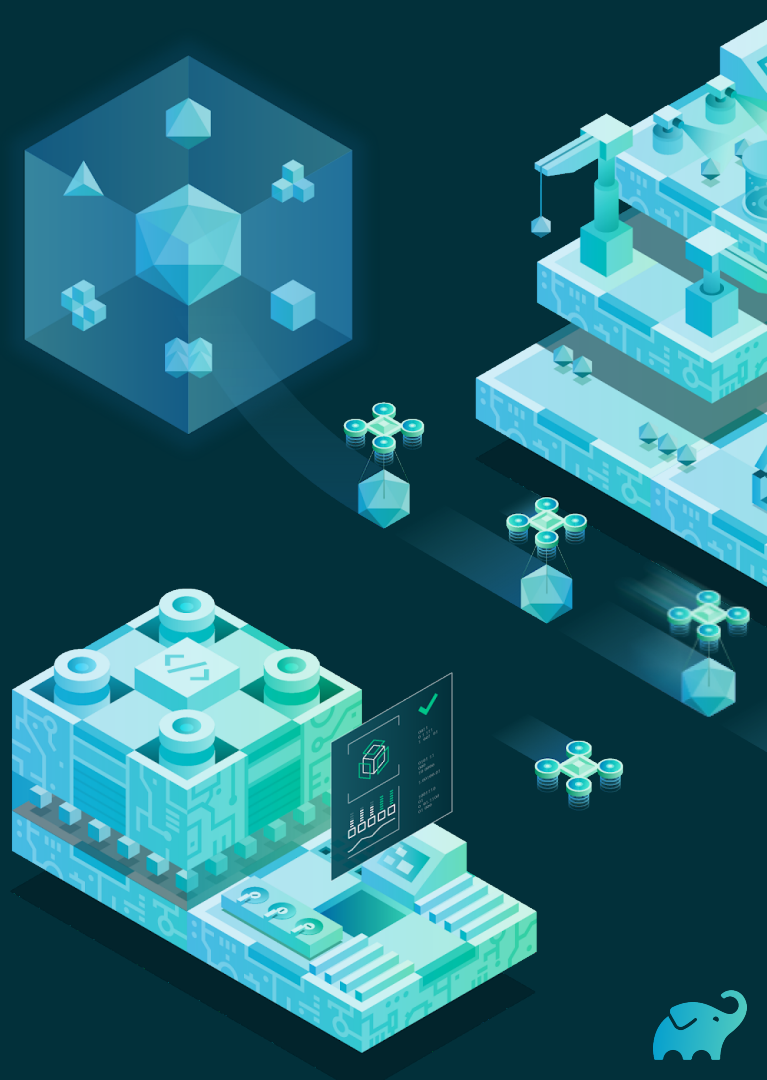
Real world dependency graph (Java)

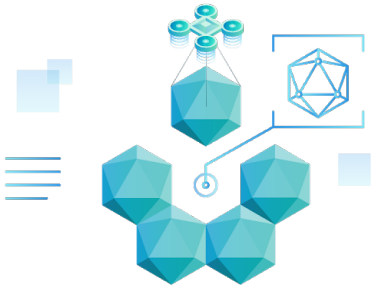


A confusing dependency

- Android library for audio recording
  - Thousands of stars on GitHub
  - Available through JitPack
- Version that ends up on the app is different
  - Executed code does not match the code on GitHub
  - Attempts network calls, sharing phone model information
- Jitpack version was shadowed by another deploy to JCenter

https://blog.autsoft.hu/a-confusing-dependency/

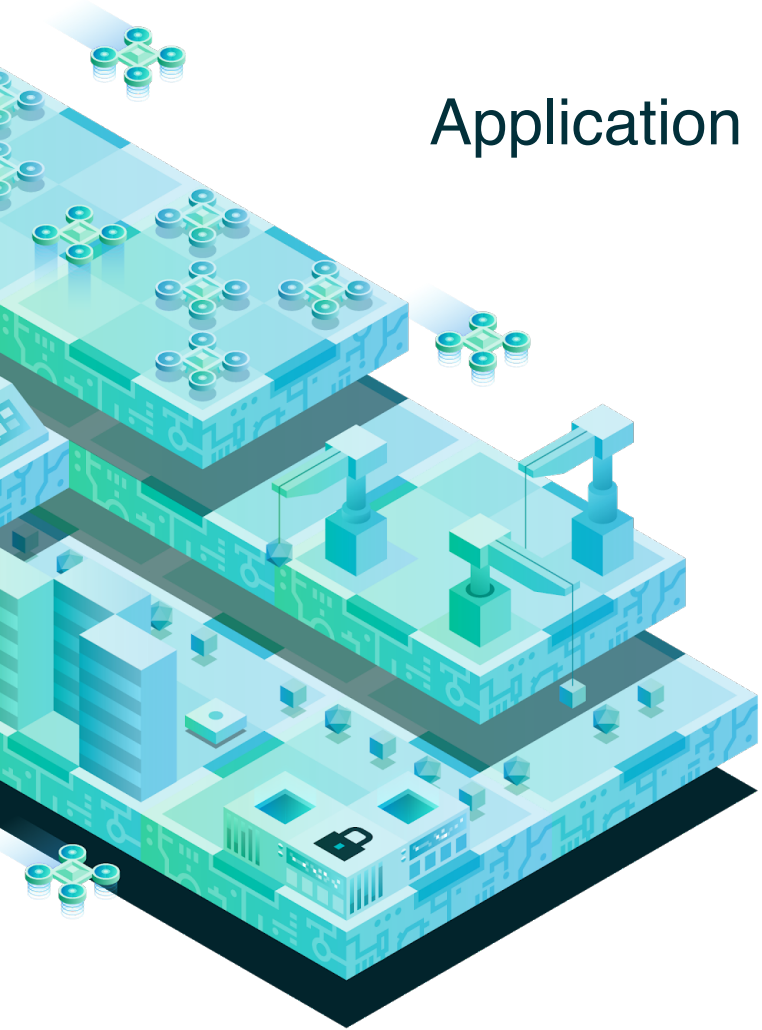




“hackers don't attack individual devices or networks directly, but rather **the companies that distribute the code used by their targets**

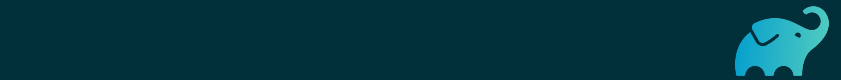
https://www.wired.com/story/supply-chain-hackers-videogames-asus-ccleaner/#

Application building: Potential attack vectors

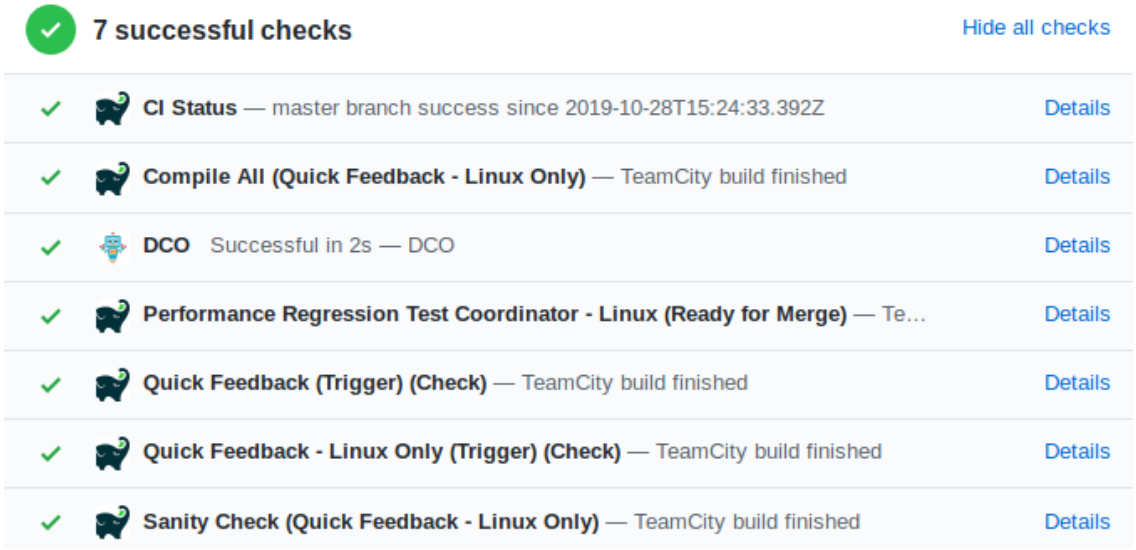


- New code
- CI infrastructure
- Dependencies
- Remote repositories
- Plugins (via plugin portal or Maven Central)
- Gradle/Maven distribution (and wrapper)
- Local file system
- Build cache / external services

New code and CI infrastructure



How to: compromise CI infrastructure



- Create a pull request
- Automatically build on CI (private or public farm)
- **Bitcoin all the things!**

Variant: Compromising OSS developer machines (example)

01 Submit a PR with compromised code (direct code, upgrade a **plugin** in the build, introduce a new one, upgrade the wrapper, edit a build script, …)

02 Developer checks out the code locally and runs **test**

03 Profit!

Never “try out” PRs

- First, look at code, **all files**
- Don’t try directly on CI
  - Or even locally!
- Be particularly picky on “obfuscated” upgrades (plugin versions, …)

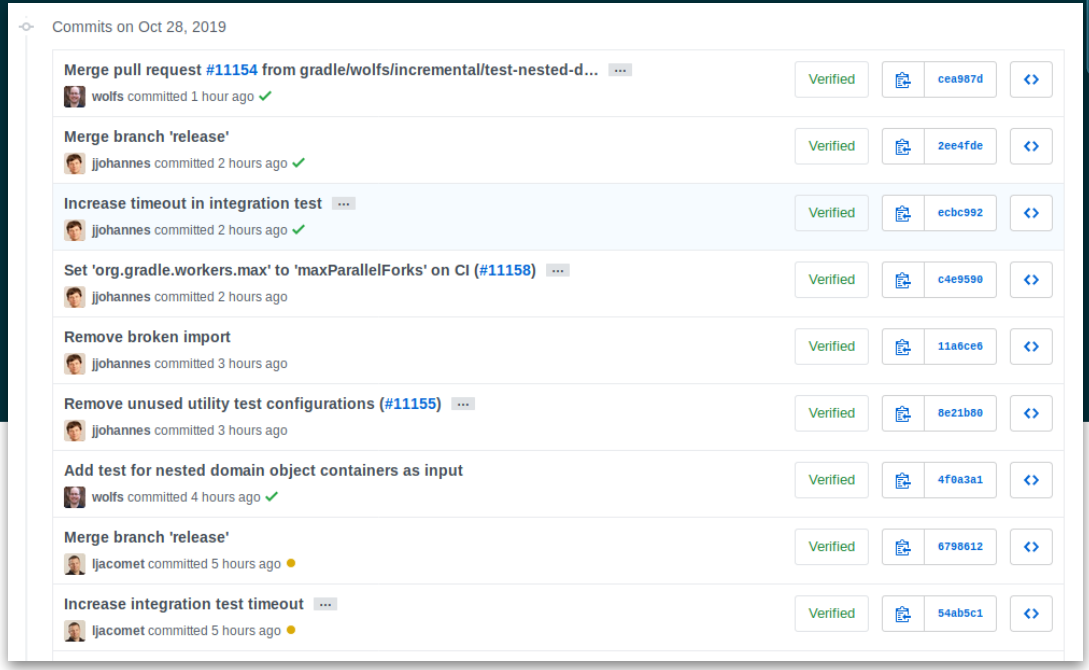
Pull request acceptance

- Use CLAs (reduces the risks)
- Perform light background verification of the author
- Review first and when you think it's ok
    - Check out the code
    - Test it
- If on CI, use **isolated**, **disposable** build agents

Signing your commits

Git lets anyone use anyone's identity
Signing proves your identity

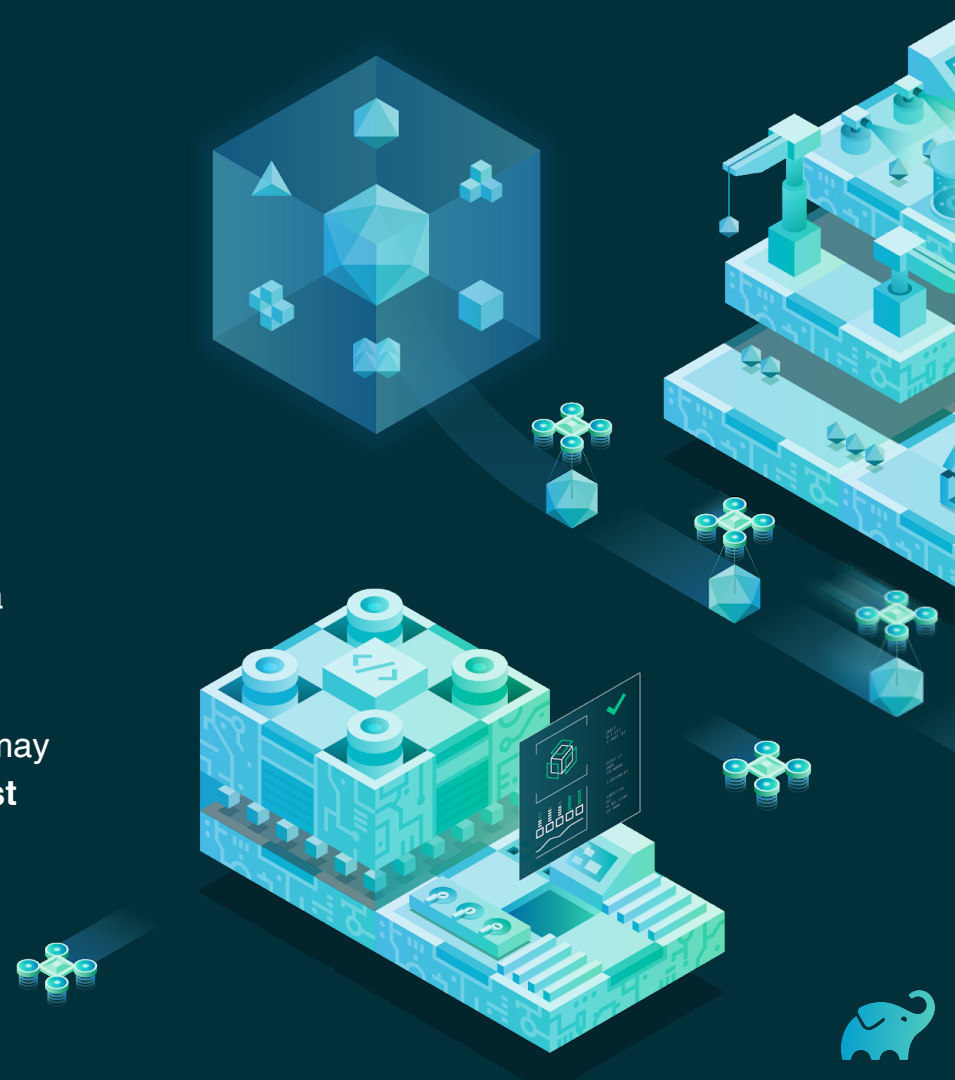
Important for legal too (who contributed what)



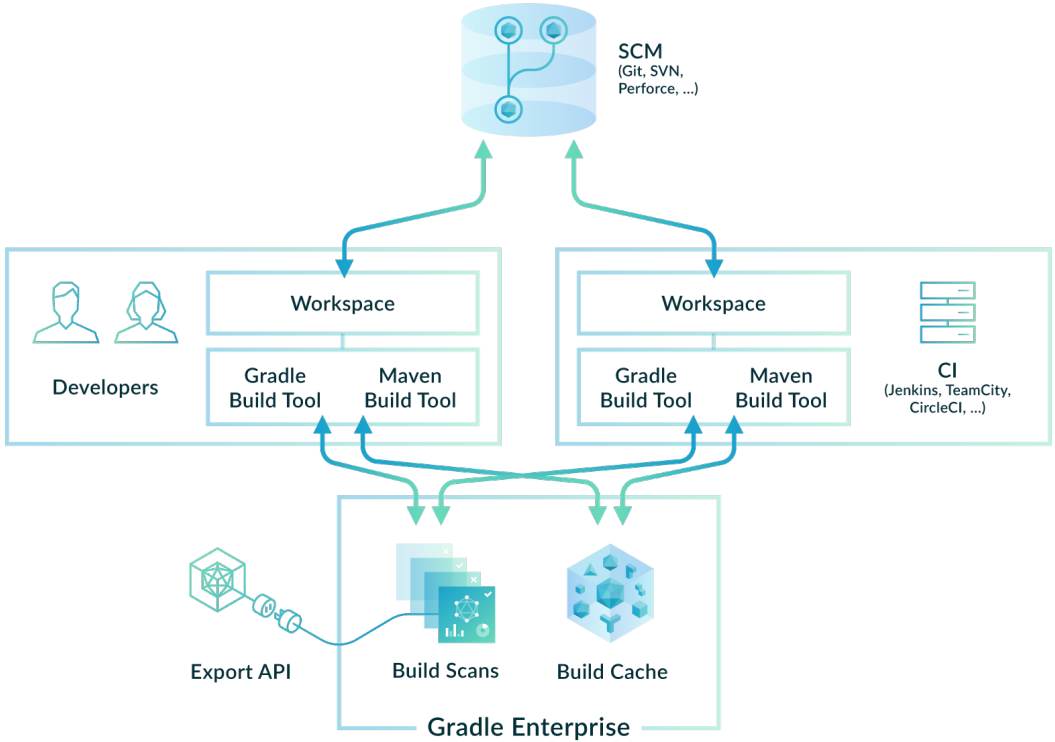
Improving CI security

Disposable containers are a good idea for security, however:

- They are **bad for performance** (extra downloads, no Gradle daemon, build bootstrapping, …)
- A single **vulnerability** in a container may be enough to **gain access to the host**



Mitigating performance issues



Mitigating performance issues

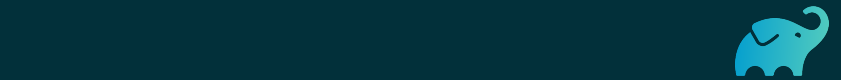
The **build cache** makes it possible to **reuse task outputs** from different build agents.

Needs **secure** connection between nodes.

Doesn’t deal with **dependency downloads**

The later can be handled by **copying or sharing** a dependency cache (Gradle 6.1 and 6.2)

Dependencies, plugins and repositories



Our Commons: Maven Central / JCenter

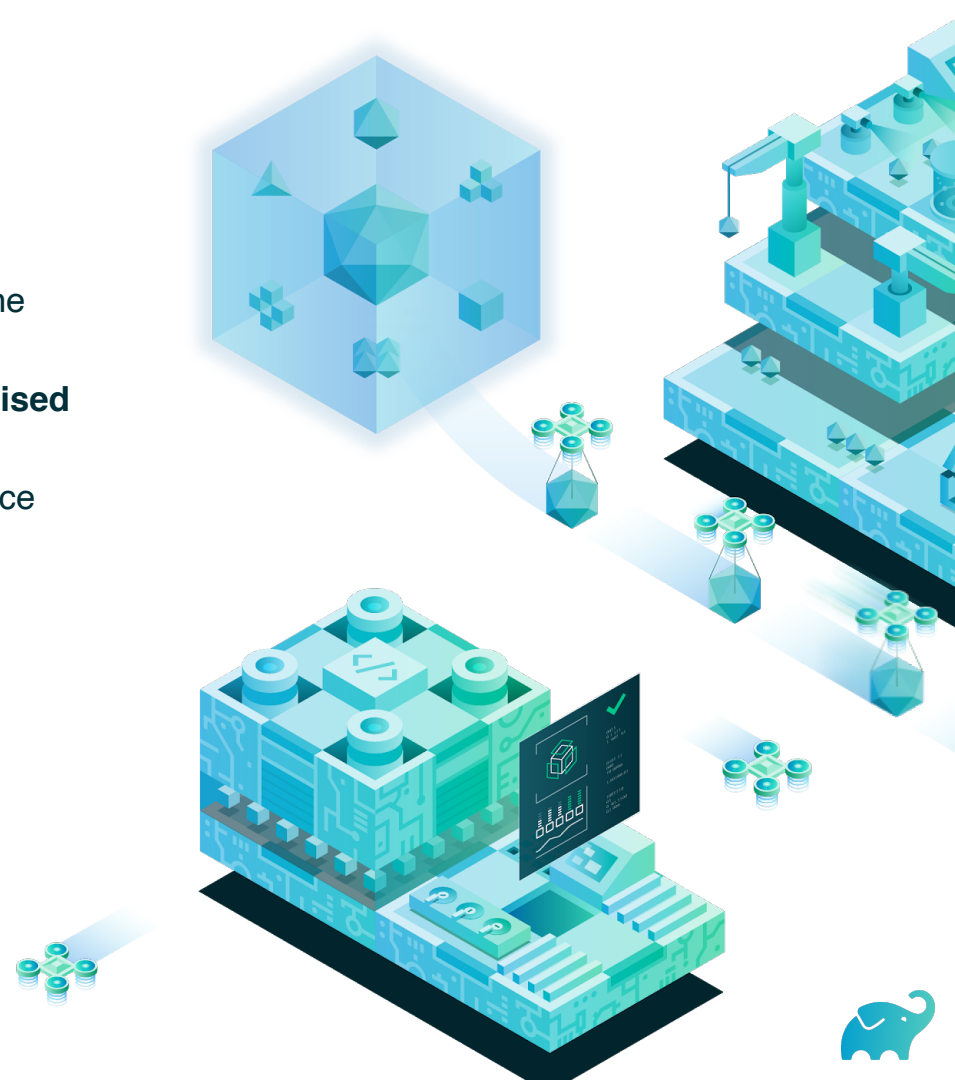
Contains millions of artifacts, mostly published as **convenience binaries**, together with:

- MD5 checksums → **unsafe for a while**
- SHA1 checksums → **really no longer safe**
- ASC signatures → **not always safe**

Checksums

- A checksum guarantees the **integrity** of the artifact (if it's safe…)
- Repository checksums **may be compromised** too!
  - Use checksums from a different source (website)
  - Publish checksums separately on a different machine!

- Gradle 6 publishes **SHA256** and **SHA512**



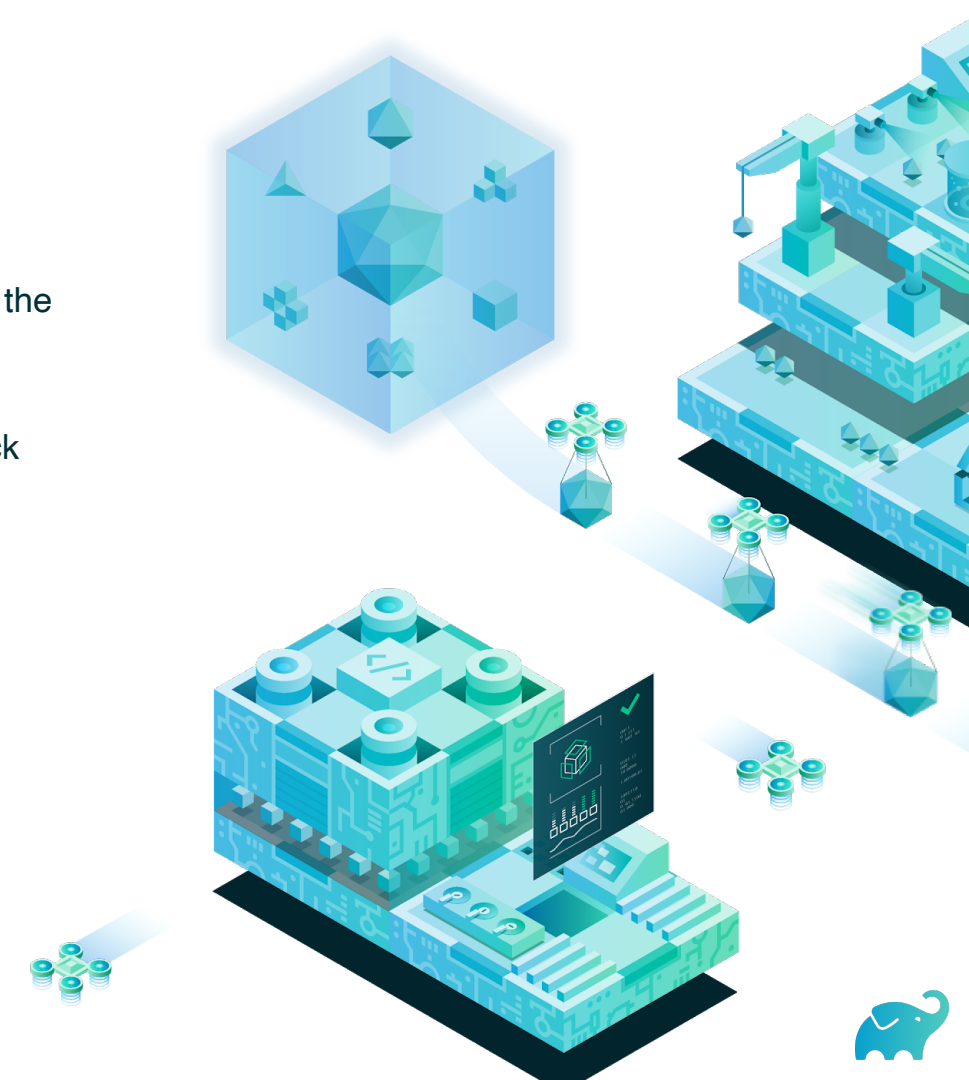
There are **broken checksums** on Maven Central.
JCenter computes SHA1/MD5 **on demand**.

Signatures

- A signature guarantees the **origin** of the artifact (if private key didn't leak)
- Commonly uses PGP
- Harder for casual developers to check

But:

- Keys sometimes lost
- Malicious authors can sign too
- ASC files use checksums too!



Verifying dependencies with Gradle

```xml
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                       xsi:schemaLocation="https://schema.gradle.org/dependency-verification https://schema.gradle.org/dependency-
   <configuration>
      <verify-metadata>false</verify-metadata>
      <verify-signatures>true</verify-signatures>
      <trusted-artifacts>
         <trust group="gradle" name="gradle"/>
         <!-- TAPI tests use a lot of different versions of the Tooling API -->
         <trust group="org.gradle" name="gradle-tooling-api"/>
         <trust group="net[.]rubygrapefruit" name="native-platform.*" version=".*snapshot.*" regex="true"/>
         <trust file=".*-javadoc[.]jar" regex="true"/>
         <trust file=".*-sources[.]jar" regex="true"/>
      </trusted-artifacts>
      <ignored-keys>
         <ignored-key id="74dafdfd6dae2441" reason="Key couldn't be downloaded from any key server"/>
      </ignored-keys>
      <trusted-keys>
         <trusted-key id="019082bc00e0324e2aef4cf00d3b328562a119a7" group="org.openjdk.jmh"/>
         <trusted-key id="0785b3eff60b1b1bea94e0bb7c25280eae63ebe5" group="org.apache.httpcomponents"/>
         <trusted-key id="07e20f0103d9dfc697c490d0368557390486f2c5" group="org.awaitility"/>
         <trusted-key id="08f0aab4d0c1a4bdde340765b341ddb020fcb6ab" group="org.bouncycastle"/>
         <trusted-key id="160a7a9cf46221a56b06ad64461a804f2609fd89" group="^com[.]github[.]shyiko($|([.].*))" regex="true"/>
         <trusted-key id="19beab2d799c020f17c69126b16698a4adf4d638" group="org.checkerframework" name="checker-qual"/>
         <trusted-key id="1fa868a348719e88b6d0de24c03ef1d7d692bcff" group="org.scala-lang"/>
         <trusted-key id="21f0942275ea159a501d375328852008a47bb995" group="^com[.]google($|([.].*))" regex="true"/>
         <trusted-key id="2db4f1ef0fa761ecc4ea935c86fdc7e2a11262cb">
            <trusting group="commons-codec"/>
            <trusting group="org.apache.commons"/>
         </trusted-key>
```

- **Supported in Gradle core** - Requires Gradle 6.2
- **Can check plugins** in addition to regular dependencies
- **Supports checking metadata** - pom.xml, .module, …
- Can be **checksum** or **signature** based
- Ability to generate first iteration of the file

Verification in Maven

- **Requires an external plugin**
- **Only cover dependencies, not plugins**
- **Doesn’t support checking metadata** - pom.xml

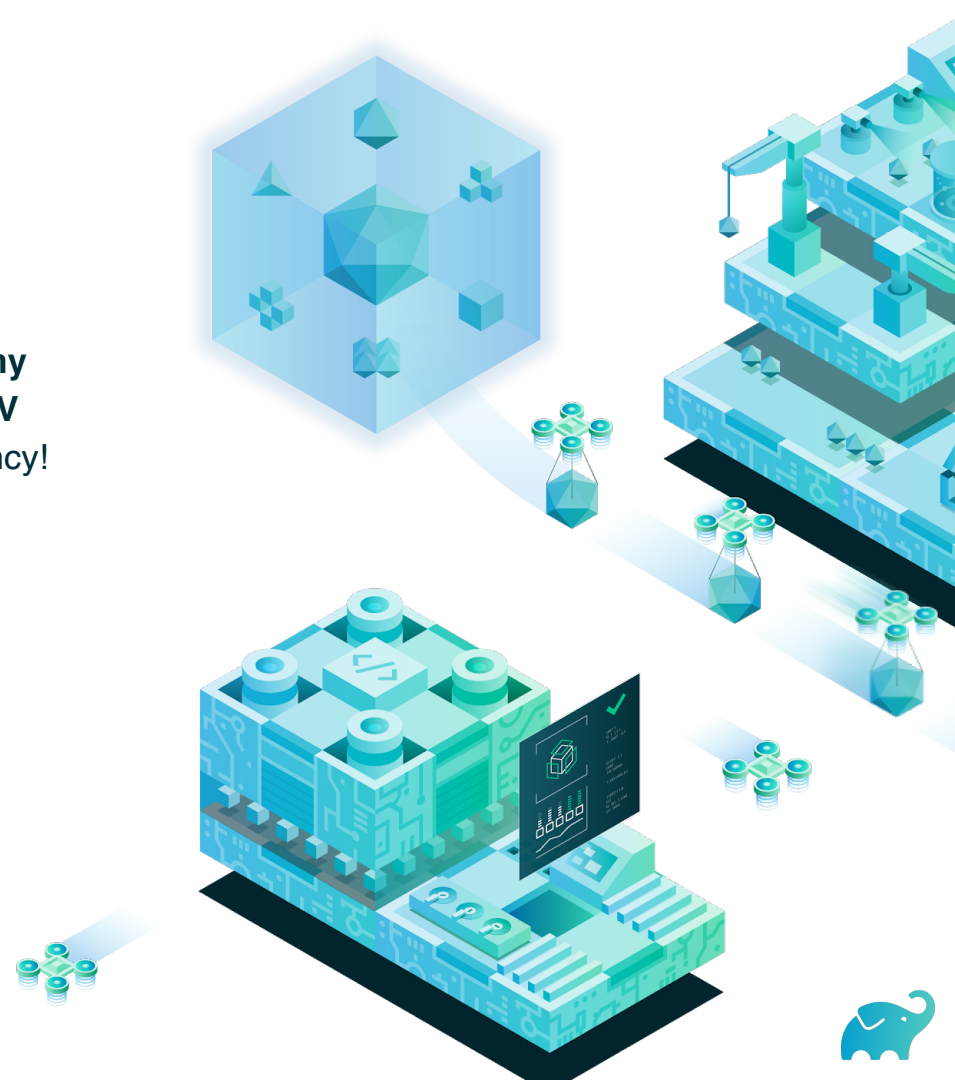
Inconsistent repositories

Different repositories may contain different **artifacts** or **metadata** for a single release!

e.g: *org.eclipse.core.runtime:3.12.0* has different dependency versions between Central and JCenter!

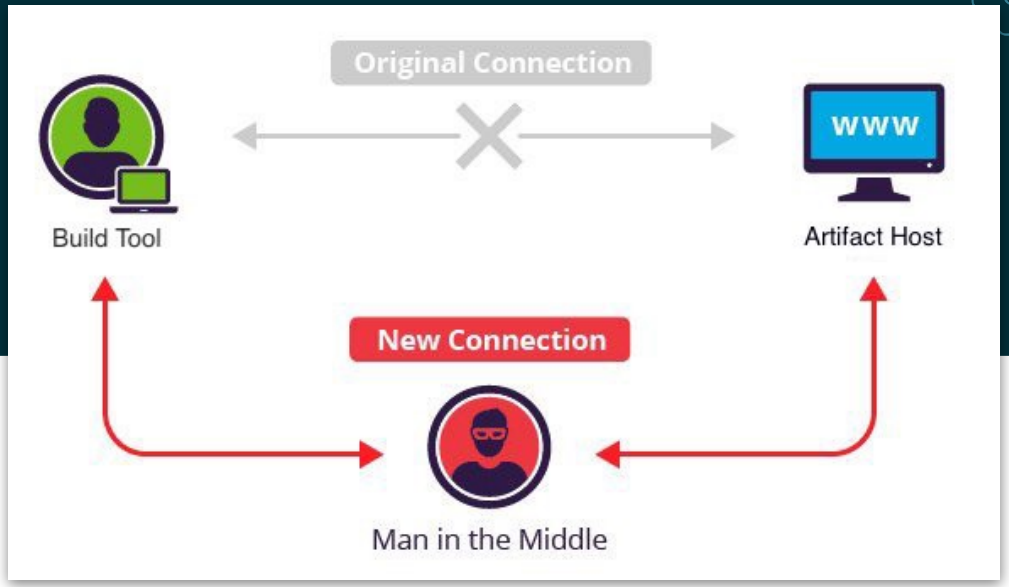
Malicious repositories

Bintray had a vulnerability which **allowed any user to publish dependencies on any GAV coordinates**, shadowing any real dependency!



Solved

Man in The Middle Attack



25% of Maven Central downloads are still using HTTP

Gradle deprecates HTTP downloads and decommissions HTTP-based services (**denied** on January 15th, 2020)

Also look at https://github.com/spring-io/nohttp

GitHub package registry

GitHub offers **custom packages publications**, effectively allowing anyone to publish any module on a **GitHub Maven repository**.

Trusting the source becomes extremely important!

(Currently requires a token to access)

Malicious repositories (Maven)

Maven uses the **declared repositories** of all dependencies during resolution.

Any repository can use the **id of an existing one** (try: central)

As a consequence it's **easy** to introduce malicious dependencies in any build!

Repository filtering with Gradle

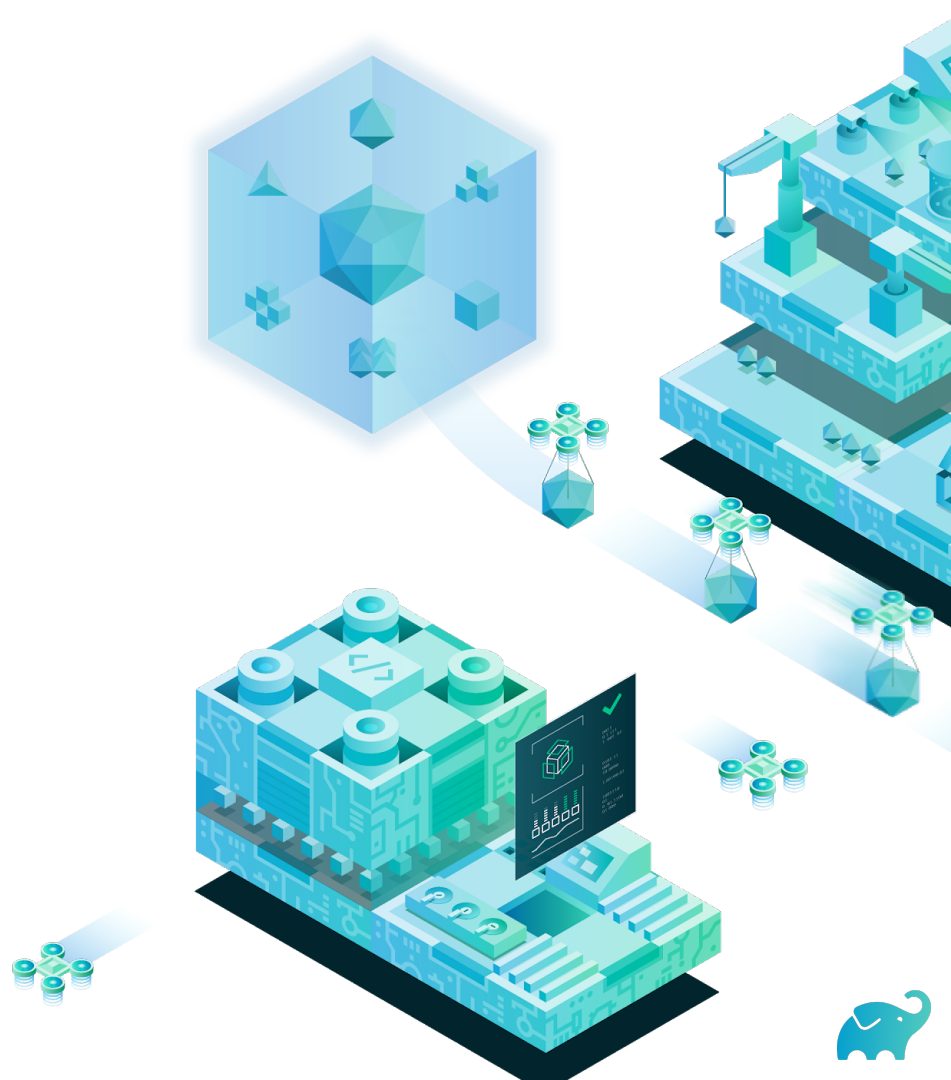
```
repositories {
    jcenter {
        content {
            includeGroup("junit")
            includeGroup("com.google.guava")
        }
    }
    maven {
        name = "myCompanyRepo"
        content {
            includeGroupByRegex("com\\.mycompany\\..*")
        }
    }
}
```

- **Know where your dependencies come from** - Precisely tell Gradle what repository contains what dependency
- **Avoid leaking details about your organization** - Avoids pinging external repositories for your internal coordinates!
- **Avoids ordering issues** - Repositories can be listed in any order if they are mutually exclusive
- **Improves performance -** No unnecessary lookups

Dealing with vulnerable dependencies

Dependency lifecycle **doesn't end at publication**:

- Bugs are discovered
- Vulnerabilities are discovered
- Bad metadata is published



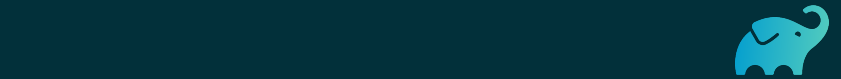
Using rich versions in Gradle

```
dependencies {
  implementation("org.apache.commons:commons-compress") {
      version {
          strictly("[1.0, 2.0[")
          prefer("1.19")
          reject("1.15", "1.16", "1.17", "1.18")
      }
      because("Versions 1.15-1.18 have a CVE")
  }
}
```

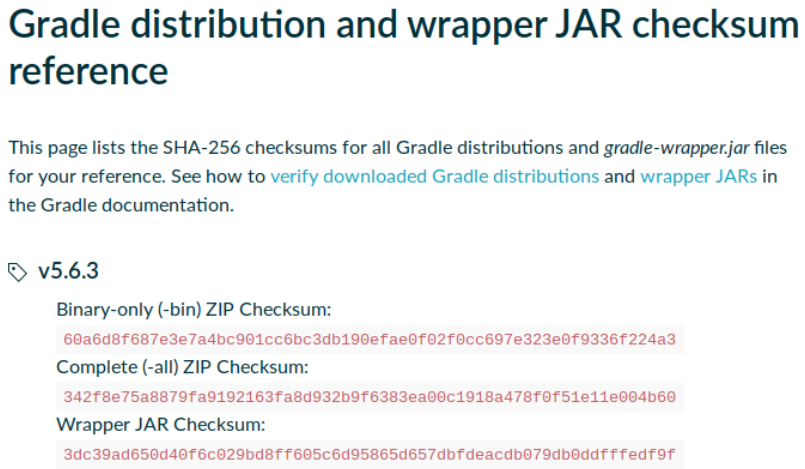
Can be added dynamically

- **Rich versions** - Allows more accurate model of why a dependency is needed
- **Graph wide** - Opinions of transitive dependencies matter
- **Allows enriching the graph with new constraints** - Consumers can tell something about transitives
- **Component metadata rules** - For amending existing metadata

Tooling



Gradle wrapper checksum verification



Gradle distribution and wrapper JAR checksum reference

This page lists the SHA-256 checksums for all Gradle distributions and *gradle-wrapper.jar* files for your reference. See how to verify downloaded Gradle distributions and wrapper JARs in the Gradle documentation.

v5.6.3

Binary-only (-bin) ZIP Checksum:

`60a6d8f687e3e7a4bc901cc6bc3db190efae0f02f0cc697e323e0f9336f224a3`

Complete (-all) ZIP Checksum:

`342f8e75a8879fa9192163fa8d932b9f6383ea00c1918a478f0f51e11e004b60`

Wrapper JAR Checksum:

`3dc39ad650d40f6c029bd8ff605c6d95865d657dbfdeacdb079db0ddfffedf9f`

In `gradle-wrapper.properties`:

```
distributionSha256Sum=371cb9fbebbe9880d147f59bab36d61eee122854ef8c9ee1ecf12b82368bcf10
```

- **Gradle wrapper will verify distribution checksums** - On every invocation
- **But you need to manually check the wrapper checksum itself** - To avoid a compromised wrapper!
- **Expected checksum is checked in** - Using a compromised distribution requires access to the source repository

Gradle wrapper GitHub action

GitHub Action

Gradle Wrapper Validation

v1 Latest version

build-test passing

Gradle Wrapper Validation Action

This action validates the checksums of Gradle Wrapper JAR files present in the source tree and fails if unknown Gradle Wrapper JAR files are found.

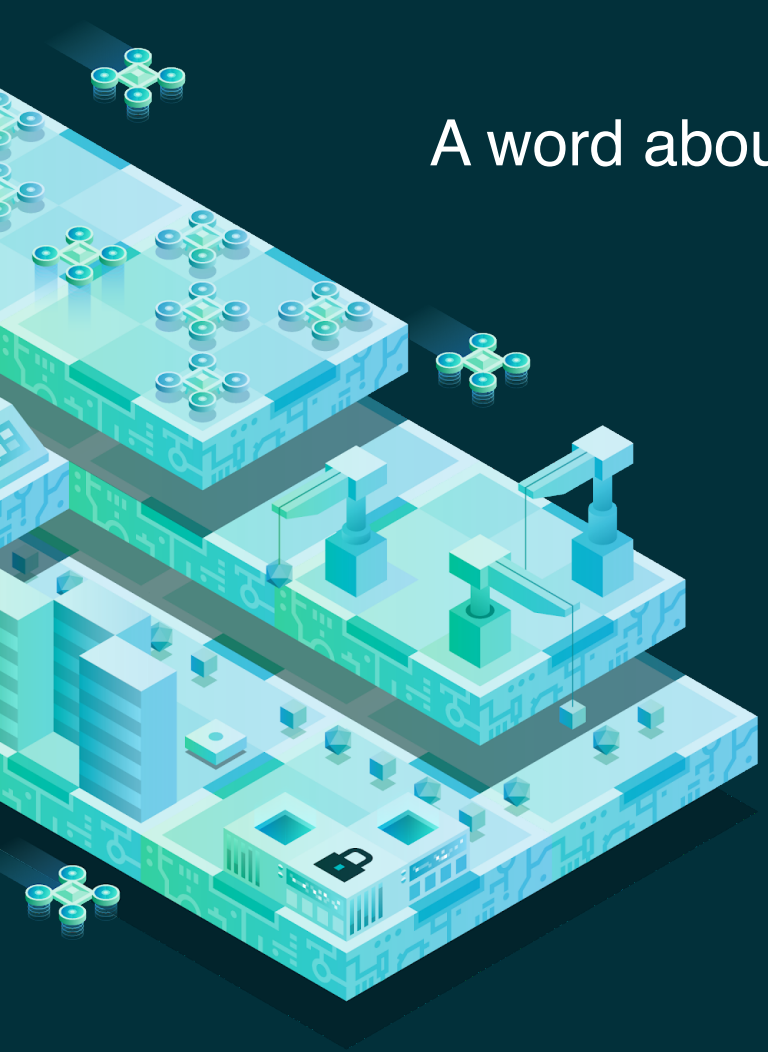
The Gradle Wrapper Problem in Open Source

The `gradle-wrapper.jar` is a binary blob of executable code that is checked into nearly 2.8 Million GitHub Repositories.

Searching across GitHub you can find many pull requests (PRs) with helpful titles like 'Update to Gradle xxx'. Many of these PRs are contributed by individuals outside of the organization maintaining the project.

Many maintainers are incredibly grateful for these kinds of contributions as it takes an item off of their backlog. We assume that most maintainers do not consider the security implications of accepting the Gradle Wrapper binary from external contributors. There is a certain amount of blind trust open source maintainers have. Further compounding the issue is that maintainers are most often greeted in these PRs with a diff to the `gradle-wrapper.jar` that looks like this.

- **GitHub action** - To be added to your workflow
- **Validates the wrapper JAR checksum itself** - To avoid a compromised wrapper!



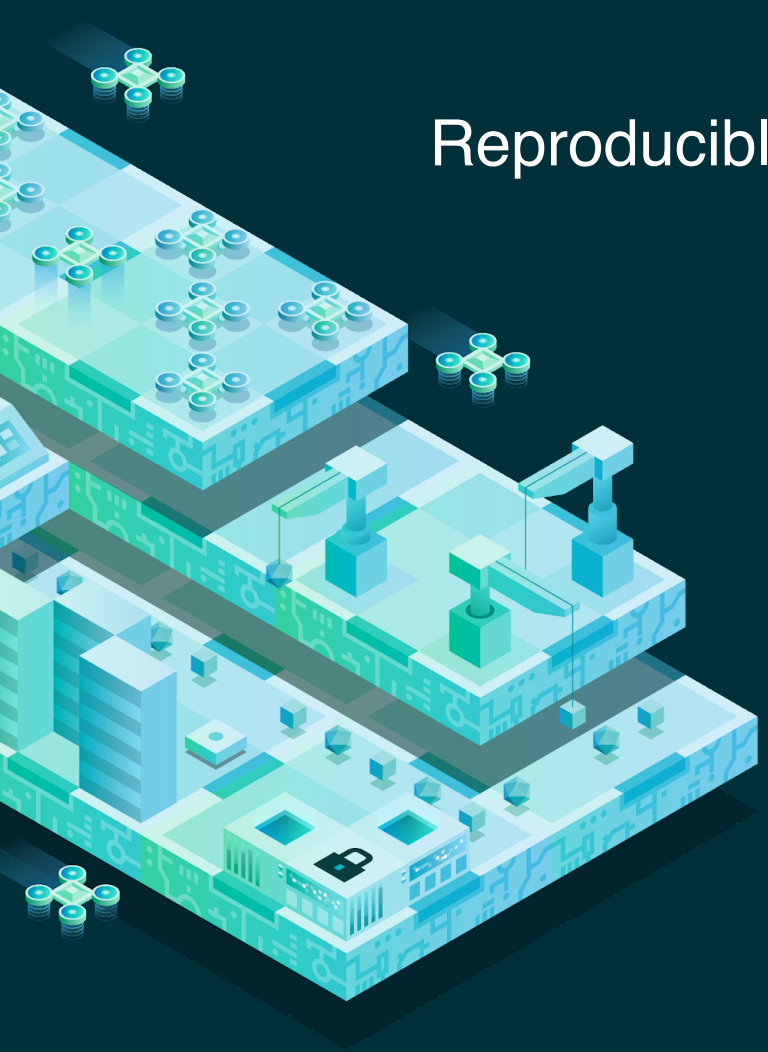
A word about 3rd-party distributions

- Gradle “official” Docker image is **not** endorsed by Gradle
- Debian and other distributions are **not** official Gradle releases
  - They use **different dependencies**
  - They build their own!
  - But they pretend to be Gradle (same version number)
- Please always prefer official releases (and Gradle wrapper if possible!)

Abusing external services

Using Gradle build cache as an *example*:

- Requires **write access** to the cache (so compromised machine or malicious employee)
- Write **custom client** to write malicious output to the cache for a known key (SHA1)
- Clients will download compromised entries



Reproducible builds

Any release should be reproducible **byte to byte**

In practice many things can go wrong:

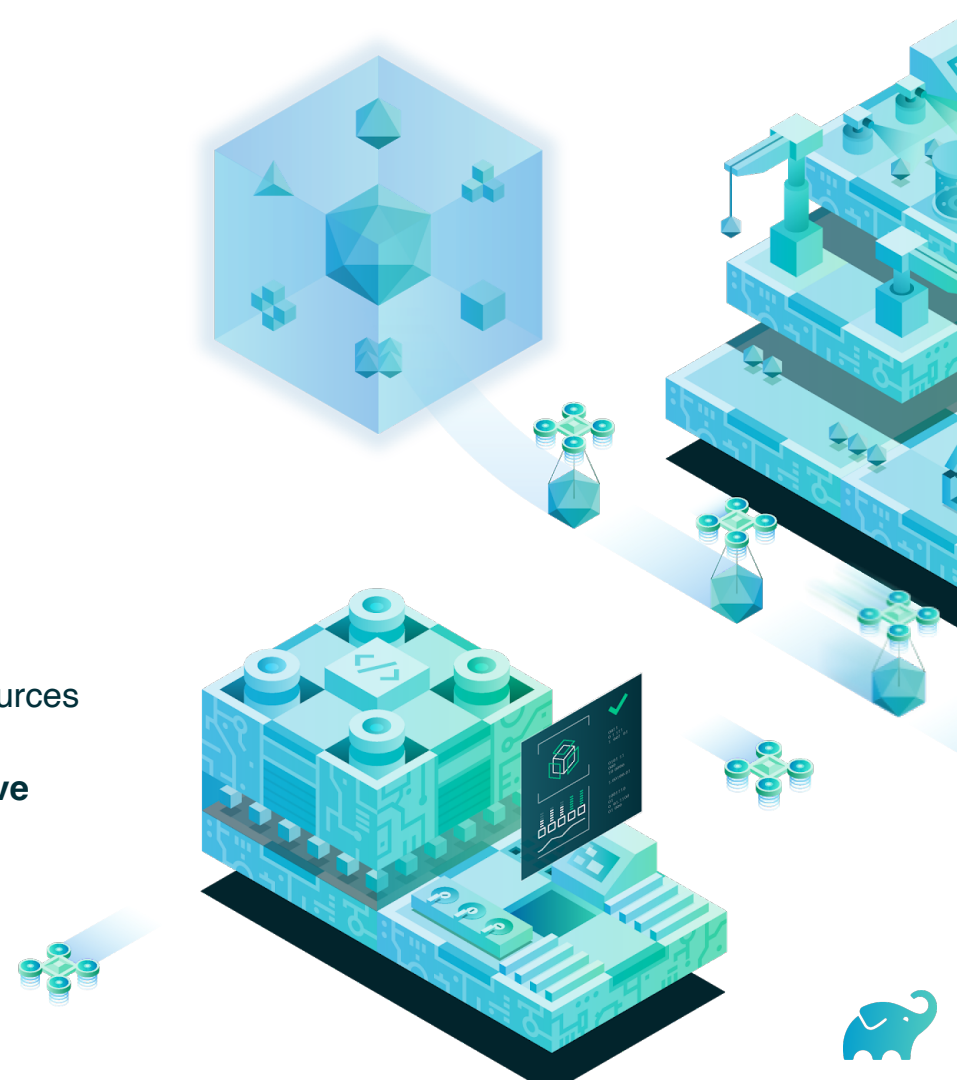
- Dynamic dependencies (ranges, 1.+, latest, …)
- Undeclared inputs
- Timestamps/debug symbols/absolute paths/...
- Dependencies removed from remote repositories
- Compiler bugs
- etc

Different approaches to reproducibility

The *Apache Software Foundation*™ way:

- Only **sources** matter
- Binaries (zip, jar, …) on Central or dist.apache.org are convenience
- Trusting requires you to build from sources

Bootstrapping problem: what about **transitive dependencies**?

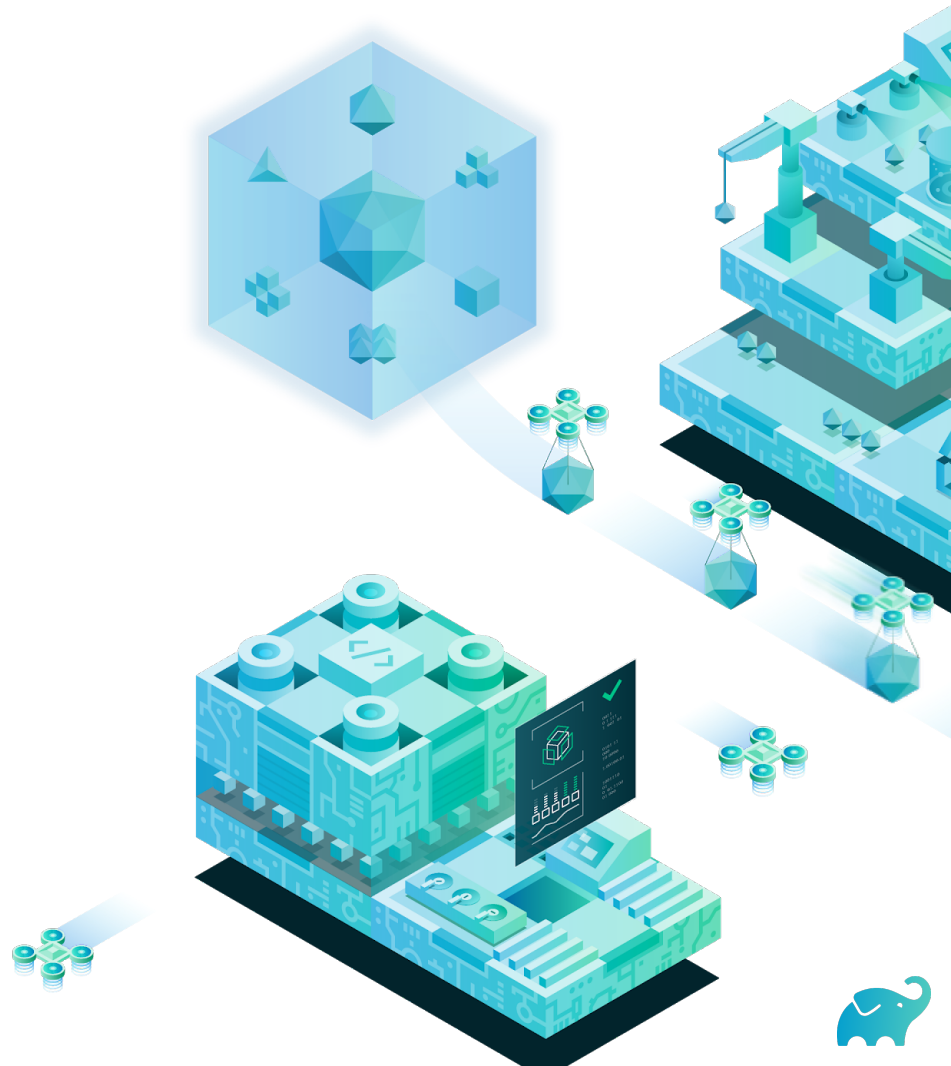


Different approaches to reproducibility

The *Google* way:

- Only **sources** matter
- No binaries, ever
- Single mono-repository

What about reuse?



We have to make compromises

If multiple organizations can build the same binaries, byte to byte, from the same sources:

- Reinforces trust
- Improves build quality
- Makes it harder to compromise

A set of best practices:

See https://reproducible-builds.org/

- **Dependency locking** - Make sure you can reuse the same versions later
- **Checksum verification** - Binaries will not be compromised
- **Reproducible archives** - Avoid timestamps, consistent ordering of archive entries, ...

Thank you!

Gradle: https://www.gradle.org

@ljacomet

References

- Small world with high risks: a study of security threats in the npm ecosystem
- Want to take over the Java ecosystem? All you need is a MITM!
- The NPM package that walked away with all your passwords
- A Post-Mortem of the Malicious event-stream backdoor
- Backdoor code found in 11 Ruby libraries
- ESlint Postmortem for Malicious Packages Published on July 12th, 2018
- Inside the Unnerving CCleaner Supply Chain Attack
- https://blog.autsoft.hu/a-confusing-dependency/