Gradle (https://docs.gradle.org)

Verifying Dependencies

version 9.8.2

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Reject Accept

On this Page

- Understanding Verification Methods
  - Checksums
  - Signatures
  - Trusted Keys and Key Servers
- Enabling Dependency Verification
  - The Verification Metadata File
  - Scope of Dependency Verification
- Disabling Dependency Verification
  - Disabling Verification for Specific Configurations
  - Skipping Javadocs and Sources
- Verification Modes
- Configuring the Console Output
- Generating and Bootstrapping Verification Metadata
  - Generating Checksums Only
  - Generating Checksums and Signatures
  - Example Verification Metadata File
  - Dry-Run Mode
- Updating Verification Metadata
  - Updating Verification Metadata Automatically
  - Updating Verification Metadata Manually
  - Configuring Key Servers, Keyrings, and Keys
- Maintaining Verification Metadata
  - Cleaning up the Verification File
  - Refreshing Missing Keys
  - Adding Keys to the Keyring
  - Exporting Keys for Faster Verification
- Understanding Signature Verification

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Working with external dependencies and plugins from third-party repositories exposes your build to significant supply chain risks. Dependencies are the most commonly attacked part of the software supply chain, and every artifact you consume, including transitively pulled-in binaries, needs to be both legitimate and unchanged.

To illustrate the risk, consider building an application that uses the `trusty-lib:1.0` library from a public repository. If an attacker successfully replaces `trusty-lib` in the repository with a malicious version having the same coordinates (`trusty-lib:1.0`), your next build will download the compromised code without any warning.

To mitigate these risks and avoid integrating compromised artifacts, Gradle provides **dependency verification**.

But dependency verification adds overhead: **So why keep it on?**

Because dependency verification is about **trust**: trust that what you download is what you think it is, and that what you ship is what you intended.

Without verification, it is easier for attackers to compromise your supply chain by publishing malicious artifacts under trusted coordinates or by tampering with artifacts in transit or at rest. Dependency verification helps protect you from these attacks by requiring you to confirm that the artifacts used in your build are the ones you expect.

Finding the right balance between security and convenience is hard, but Gradle provides options to let you choose the right level for your needs.

NOTE

Dependency verification is meant to protect yourself from compromised dependencies, not to prevent you from including *vulnerable* dependencies.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Understanding Verification Methods

Gradle provides two complementary mechanisms for dependency verification:

1. **Checksums** – Verify the **integrity** of the artifact (that file contents haven't changed)
2. **Signatures** – Verify the **provenance** of the artifact (who published it)

Used together, they help ensure you get the right library or plugin from a trustworthy author. Checksums alone verify integrity but not authenticity. Signatures alone verify authenticity but may use weak hashes. You can use signatures whenever they are available, and fall back to checksums when artifacts are not signed. **But for the strongest security, use both.**

WARNING

Signatures can also be used to assess integrity like checksums, but signatures are an *encrypted hash* of the artifact contents and some metadata, not artifacts themselves. If the signature uses an unsafe hash (even `SHA-1`), you're not correctly assessing file integrity. For this reason, if you care about both integrity and authenticity, you need to add both signatures *and* checksums to your verification metadata.

Checksums

A checksum is a short **fingerprint** of a file's contents.

To produce a checksum, the library or plugin file is passed through a cryptographic hash function (for example, `SHA-256`), which returns a fixed-length string that uniquely represents that exact content. If the file changes in any way, the checksum will also change.

With **checksums**, the build tool keeps a list of expected hashes for each artifact:

- You record the expected checksum (for example, `SHA-256`) of a library or plugin in the `gradle/verification-metadata.xml` file.
- When the build downloads that artifact, it computes the checksum of the downloaded bytes.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```xml
<artifact name="trusty-lib-1.0.jar">
    <sha256 value="5e87d84077de0c60d9ff4eedbc9fac1506693019eb6f55652afc9e1ff235bd15"/>
</artifact>
```

Signatures

A signature is a cryptographic **stamp** that proves authenticity.

To produce a signature, the author uses their **private key** to create a digital signature over the library or plugin.

With **signatures**, you verify **who** produced the artifact, not just that it hasn't changed:

- The author signs the checksum of the artifact's file contents along with additional metadata using their private key.
- You verify and add trusted public keys to the `gradle/verification-metadata.xml` file.
- When the artifact is downloaded, the build verifies the signature against the trusted public key.
- If the signature is missing, invalid, or signed by an untrusted key, the build fails.

For example, the combined checksum and signature entry for `trusty-lib-1.0.jar` might look like this:

verification-metadata.xml

```xml
<artifact name="trusty-lib-1.0.jar">
    <sha256 value="5e87d84077de0c60d9ff4eedbc9fac1506693019eb6f55652afc9e1ff235bd15"/>
    <pgp value="D4C89EA4AAF455FD88B22087EFE8086F9E93774E"/>
</artifact>
```

Trusted Keys and Key Servers

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```xml
<trusted-keys>
    <trusted-key
        id="06D34ED6FF73DE368A772A781063FE98BCECB758"
        group="com.puppycrawl.tools"
        name="checkstyle"/>
</trusted-keys>
```

Public keys are typically fetched from key servers, which are repositories for public keys, such as:

- https://keyserver.ubuntu.com
- https://keys.openpgp.org
- https://pgp.mit.edu

Gradle uses these trusted keys plus the checksum/signature data in `verification-metadata.xml` to ensure that dependencies come from the expected author and have not been tampered with.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Enabling Dependency Verification

To enable dependency verification, you need to create a verification metadata file that tells Gradle which checksums and signatures to verify. Once this file exists, Gradle automatically verifies all dependencies during every build.

The Verification Metadata File

Dependency verification is automatically enabled as soon as the verification metadata file is detected. The file must be located at `$PROJECT_ROOT/gradle/verification-metadata.xml`.

NOTE

Currently, the only source of dependency verification metadata is this XML configuration file. Future versions of Gradle may include other sources (for example, via external services).

A minimal configuration consists of the following:

```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.o
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>false</verify-signatures>
    </configuration>
</verification-metadata>
```

With this configuration, Gradle verifies all artifacts using checksums but will not verify signatures.

Gradle verifies any artifact downloaded via its dependency management engine, including but not limited to:

- Artifact files (e.g., JAR files, ZIPs) used during the build

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//). By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Gradle does *not* verify changing dependencies (specifically `SNAPSHOT` versions) or locally produced artifacts (such as JARs generated during the build itself). By nature, the checksums and signatures of these files change constantly.

WARNING

Using such a minimal configuration file will cause a project with external dependencies or plugins to fail immediately, as the file does not yet contain the required checksums for verification. See Generating and Bootstrapping Verification Metadata to learn more.

Scope of Dependency Verification

The dependency verification configuration is *global*; a single file is used to verify the entire build. Specifically, the same file applies to all subprojects, the root project, and `buildSrc`.

When using an included build, the following rules apply:

- The configuration file of the *current* build is used for verification.
- If an included build has its own verification metadata, that configuration is ignored in favor of the *current* build's settings.
- Much like upgrading a dependency, including a new build may require you to update your *current* verification metadata.

Because this configuration is global and strict, the most efficient way to begin is by generating a baseline configuration for your existing build.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Disabling Dependency Verification

By default, Gradle verifies both artifacts (JARs, ZIPs, etc.) and their metadata files (POM files, Ivy descriptors, Gradle Module Metadata). Metadata verification can significantly increase the size of your configuration file.

To disable metadata verification, set the `<verify-metadata>` flag to `false`:

```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.o
   <configuration>
      <verify-metadata>false</verify-metadata>
      <verify-signatures>false</verify-signatures>
    </configuration>
</verification-metadata>
```

WARNING

Disabling metadata verification reduces security. Compromised metadata files could introduce malicious transitive dependencies into your build.

Disabling Verification for Specific Configurations

Dependency verification is enabled globally to provide the strongest security level possible. However, plugins may need to resolve additional dependencies where verification doesn't make sense.

For example, a plugin might check for newer versions of a library. It doesn't make sense to require checksums for versions you don't know about yet. In these cases, you can disable verification for specific configurations using the API.

WARNING

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Kotlin Groovy

build.gradle

```groovy
configurations {
    myConfiguration {
        resolutionStrategy {
            disableDependencyVerification()
        }
    }
}
```

You can also disable verification on detached configurations:

Kotlin Groovy

```groovy
def detached = configurations.detachedConfiguration(dependencies.create("group:name:1.0"))
detached.resolutionStrategy.disableDependencyVerification()
```

Skipping Javadocs and Sources

By default, Gradle verifies all downloaded artifacts, including javadocs and sources. This can cause issues with IDEs that automatically download them during import.

To avoid this, you can configure Gradle to automatically trust all javadocs and sources:

```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.o
   <configuration>
      <trusted-artifacts>
         <trust file=".*-javadoc[.]jar" regex="true"/>
```

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

NOTE

Trusting javadoc and source artifacts means they could be tampered with without detection.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Verification Modes

Dependency verification can be expensive, or sometimes verification gets in the way of day-to-day development (frequent dependency upgrades, for example). You might want to enable verification on CI servers but not on local machines.

Gradle provides three verification modes:

Mode	Description
`strict` (default)	Verification fails as early as possible to avoid using compromised dependencies during the build
`lenient`	Runs the build even if there are verification failures. Verification errors are displayed without causing build failure
`off`	Verification is completely ignored

You can activate these modes on the CLI using the `--dependency-verification` flag:

```
BASH
$ ./gradlew --dependency-verification lenient build
```

Alternatively, set the `org.gradle.dependency.verification` system property on the CLI:

```
BASH
$ ./gradlew -Dorg.gradle.dependency.verification=lenient build
```

Or in a `gradle.properties` file:

```
PROPERTIES
org.gradle.dependency.verification=lenient
```

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Configuring the Console Output

By default, if dependency verification fails, Gradle generates a small summary about the verification failure as well as an HTML report containing the full information about the failures.

If your environment prevents you from reading this HTML report file (for example, if you run a build on CI, and it's not easy to fetch the remote artifacts), Gradle provides a way to opt in to a verbose console report.

Add this Gradle property to your `gradle.properties` file:

```properties
org.gradle.dependency.verification.console=verbose
```

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Generating and Bootstrapping Verification Metadata

Gradle can automatically generate a dependency verification file by downloading all your dependencies and recording their checksums or signatures. This is called **bootstrapping**.

However, bootstrapping has a critical security limitation: **it trusts whatever is currently in your repositories**. If a malicious dependency has already been introduced into your build (or into a repository you use), Gradle will record the compromised artifact's checksum or signature.

This is why you must **review the generated verification file**. Bootstrapping is convenient for getting started or updating your verification file, but it's not a substitute for manual verification of critical dependencies.

Generating Checksums Only

To enable checksum verification, you need to generate a configuration file based on your current, trusted dependencies.

Run the following command in your terminal:

```bash
$ ./gradlew --write-verification-metadata sha256
```

What this does:

1. It scans all the dependencies in your project.
2. It calculates the `SHA-256` checksum for each.
3. It creates a file located at `gradle/verification-metadata.xml`.

Gradle supports `MD5`, `SHA1`, `SHA-256`, and `SHA-512` checksums. However, only `SHA-256` and `SHA-512` are considered secure for modern use. `SHA-1` has known collisions and `MD5` should not be used.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//). By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Gradle will:

- Resolve your dependencies.
- Download and/or generate the required verification information.
- Write it in a VCS-friendly XML file at `gradle/verification-metadata.xml`.

The `write-verification-metadata` flag requires the list of checksums that you want to generate or `pgp` for signatures.

Executing this command will cause Gradle to:

- Resolve all resolvable configurations, which includes:
  - Configurations from the root project
  - Configurations from all subprojects
  - Configurations from `buildSrc`
  - Included builds configurations
  - Configurations used by plugins
- Download all artifacts discovered during resolution
- Compute the requested checksums and possibly verify signatures depending on what you asked
- At the end of the build, generate the configuration file which will contain the inferred *verification metadata*

Gradle only supports verification of signatures published as ASCII-armored PGP files (`.asc`) on remote repositories. Not all artifacts are published with signatures, and signature verification doesn't guarantee the signatory was legitimate—only that the artifact was signed by a specific key.

Example Verification Metadata File

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```xml
<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

    <components>

        <!-- BOTH SIGNATURE AND CHECKSUM -->
        <component group="com.google.guava" name="failureaccess" version="1.0.3">
            <artifact name="failureaccess-1.0.3.jar">
                <pgp value="BDB5FA4FE719D787FB3D3197F6D4A1D411E9D1AE"/>
                <sha256 value="cbfc3906b19b8f55dd7cfd6dfe0aa4532e834250d7f080bd8d211a3e246b59cb"
                        origin="Verified"
                        reason="Added manually to fix CI"/>
            </artifact>
        </component>

        <!-- CHECKSUM ONLY -->
        <component group="antlr" name="antlr" version="2.7.7">
            <artifact name="antlr-2.7.7.jar">
                <sha256 value="88fbda4b912596b9f56e8e12e580cc954bacfb51776ecfddd3e18fc1cf56dc4c"
                        origin="Verified"
                        reason="Artifact is not signed"/>
            </artifact>
        </component>

        <!-- SIGNATURE ONLY -->
        <component group="com.beust" name="jcommander" version="1.78">
            <artifact name="jcommander-1.78.jar">
                <pgp value="C70B844F002F21F6D2B9C87522E44AC0622B91C3"/>
                <pgp value="DCBA03381EF6C89096ACD985AC5EC74981F9CDA6"/>
            </artifact>
        </component>

    </components>
</verification-metadata>
```

Dry-Run Mode

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

To preview changes, add the `--dry-run` flag:

```
BASH
$ ./gradlew --write-verification-metadata sha256 help --dry-run
```

Instead of generating the `verification-metadata.xml` file, Gradle will create a new file called `verification-metadata.dryrun.xml` that you can review.

NOTE

Because `--dry-run` doesn't execute tasks, it runs much faster but will miss any dependency resolution that happens at task execution time.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Updating Verification Metadata

Once the `verification-metadata.xml` file is generated, every time a developer or CI server runs the build, Gradle will compare the downloaded file's checksum against the one in the XML file.

When you add a new library or update a version, the build will fail because the new checksum isn't in your XML file yet.

You have two options for updating the verification metadata: auto-update or manual update.

Updating Verification Metadata Automatically

Run the `--write-verification-metadata` command again to append the new hashes:

```
BASH
$ ./gradlew --write-verification-metadata sha256
```

Bootstrapping can either be used to create the file from the beginning or to *update* an existing file with new information. Therefore, it's recommended to always use the same parameters once you started bootstrapping.

Generation is incremental and preserves existing entries. Gradle-generated checksums will have an `origin` attribute starting with `"Generated by Gradle"` to indicate they need review. Manually added entries and header comments are preserved, making it easy to update the file for new dependency versions by regenerating and reviewing the diff.

verification-metadata.xml

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```xml
<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

    <components>
      <component group="aopalliance" name="aopalliance" version="1.0">
         <artifact name="aopalliance-1.0.jar">
            <sha256 value="0addec670fedcd3f113c5c8091d783280d23f75e3acb841b61a9cdb079376a08" origin
         </artifact>
         <artifact name="aopalliance-1.0.pom">
            <sha256 value="26e82330157d6b844b67a8064945e206581e772977183e3e31fec6058aa9a59b" origin
         </artifact>
      </component>
    </components>
</verification-metadata>
```

After regenerating the file, follow these steps:

1. **Review the diff** - Check what changed in your version control system.
2. **Remove unnecessary entries** - Delete entries for old dependency versions that are no longer used.
3. **Verify new entries** - Ensure new checksums are correct by checking against official sources when possible.
4. **Handle ignored keys** - If any ignored keys were added, attempt to download them from a keyserver and add them to the keyring if found. See adding keys to the keyring.
5. **Update all "Generated by Gradle" origins** - You must review and update all entries with `origin="Generated by Gradle"`. Common reasons for generation include:
   - Artifact is not signed
   - PGP signature verification failed
   - Key couldn't be downloaded

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

WARNING

There are dependencies that Gradle *cannot* discover automatically. In particular, dependencies that are only resolved during task execution or in custom dependency resolution logic may not be included in the generated file. You may need to add these manually.

Updating Verification Metadata Manually

You can add a new `<component>` entry for a checksum or signature to the XML file manually.

Adding Checksums for an Artifact

External components are identified by GAV coordinates (`group`, artifact `name`, `version`), and each artifact by its file name.

For example, to add verification for Apache PDFBox (https://pdfbox.apache.org) with GAV coordinates:

- group: `org.apache.pdfbox`
- name: `pdfbox`
- version: `2.0.17`

Using this dependency triggers the download of two files:

- `pdfbox-2.0.17.jar` - the main artifact
- `pdfbox-2.0.17.pom` - the metadata file

You need to declare checksums for both (unless you disabled metadata verification):

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```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.o
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>false</verify-signatures>
   </configuration>
   <components>
      <component group="org.apache.pdfbox" name="pdfbox" version="2.0.17">
         <artifact name="pdfbox-2.0.17.jar">
            <sha512 value="7e11e54a21c395d461e59552e88b0de0ebaf1bf9d9bcacadf17b240d9bbc29bf6beb8e36
         </artifact>
         <artifact name="pdfbox-2.0.17.pom">
            <sha512 value="82de436b38faf6121d8d2e71dda06e79296fc0f7bc7aba0766728c8d306fd1b0684b5379
         </artifact>
      </component>
   </components>
</verification-metadata>
```

Where to Get Checksums

Checksums are generally published alongside artifacts on public repositories. However, if a dependency is compromised in a repository, its checksum likely is too.

It's a good practice to get checksums from a different source than where artifacts are hosted (typically the library's official website). It's much harder to compromise both the repository and the official website simultaneously.

In the example above, the checksum for the JAR was found on the official PDFBox website, while the POM checksum was verified manually. The `origin` attribute documents where the checksum came from, helping others understand how trustworthy the verification is.

Using Multiple Checksums

You can declare multiple checksums for the same artifact. Gradle will verify *all* of them and fail if

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```xml
<component group="org.apache.pdfbox" name="pdfbox" version="2.0.17">
   <artifact name="pdfbox-2.0.17.jar">
      <md5 value="c713a8e252d0add65e9282b151adf6b4" origin="official site"/>
      <sha1 value="b5c8dff799bd967c70ccae75e6972327ae640d35" origin="official site"/>
   </artifact>
</component>
```

Use multiple checksums when:

- The official site only publishes insecure checksums (MD5, SHA-1). While each can be faked individually, faking multiple checksums for the same artifact is harder.
- You want to add generated checksums alongside manually verified ones for extra validation.
- You're upgrading to more secure checksums but don't want to remove existing ones yet.

Handling Transitive Dependencies

Using a dependency like `pdfbox` will bring in transitive dependencies. If your verification file only includes checksums for direct dependencies, the build will fail:

```text
Execution failed for task ':compileJava'.
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact commons-logging-1.2.jar (commons-logging:commons-logging:1.2) in repository 'Mave
    - On artifact commons-logging-1.2.pom (commons-logging:commons-logging:1.2) in repository 'Mave
```

This indicates you need to add checksums for `commons-logging` and all other transitive dependencies. See Troubleshooting Verification Failures for more details.

Trusting Multiple Checksums

It's common to have different checksums for the same artifact in different repositories. This can happen because:

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```xml
<component group="org.apache" name="apache" version="13">
   <artifact name="apache-13.pom">
      <sha256 value="2fafa38abefe1b40283016f506ba9e844bfcf18713497284264166a5dbf4b95e">
         <also-trust value="ff513db0361fd41237bef4784968bc15aae478d4ec0a9496f811072ccaf3841d"/>
      </sha256>
   </artifact>
</component>
```

You can have as many `also-trust` entries as needed, though typically you shouldn't need more than two.

Adding Keys for an Artifact

To trust a key for signature verification, add a `pgp` entry to your verification metadata:

```xml
<component group="com.github.javaparser" name="javaparser-core" version="3.6.11">
   <artifact name="javaparser-core-3.6.11.jar">
      <pgp value="8756c4f765c9ac3cb6b85d62379ce192d401ab61"/>
   </artifact>
</component>
```

This means you trust `com.github.javaparser:javaparser-core:3.6.11` if it's signed with key `8756c4f765c9ac3cb6b85d62379ce192d401ab61`.

A `pgp` entry also accepts optional `origin` and `reason` attributes to document where the key was obtained or verified and why it is trusted:

```xml
<component group="com.github.javaparser" name="javaparser-core" version="3.6.11">
   <artifact name="javaparser-core-3.6.11.jar">
      <pgp value="8756c4f765c9ac3cb6b85d62379ce192d401ab61"
           origin="https://keyserver.ubuntu.com"
           reason="Verified against the maintainer's website"/>
```

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-core:3.6.11) in repo
```

A different failure occurs when the signing key cannot be located on any key server at all:

```
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact foo-1.0.jar (org:foo:1.0) in repository 'maven': Artifact was signed with key '14
```

The trailing parenthetical reports how many other keys you already trust for the failing artifact, distinguishing keys trusted for the specific `group:module` from keys trusted for the whole `group`. A non-zero count is a strong signal that the signing key has been rotated, rather than this being a brand-new module or group you are trusting for the first time.

WARNING

Gradle requires full fingerprint IDs (40 characters, e.g., `b801e2f8ef035068ec1139cc29579f18fa8fd93b`) for `pgp` and `trusted-key` elements, not short or long IDs. This minimizes the risk of collision attacks. For `ignored-key` elements, you can use either fingerprints or long (64-bit) IDs.

NOTE

The key IDs shown in error messages are from the signature file itself. This doesn't necessarily mean you should trust them—if a malicious entity signed the artifact, Gradle won't detect that. Always verify keys against official sources.

Adding Global Keys

Signature verification can be simplified by trusting keys at the group level instead of declaring them for each individual artifact. This is useful when the same key signs multiple artifacts from the same publisher.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```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.o
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>true</verify-signatures>
      <trusted-keys>
         <trusted-key id="8756c4f765c9ac3cb6b85d62379ce192d401ab61" group="com.github.javaparser"/>
      </trusted-keys>
   </configuration>
   <components/>
</verification-metadata>
```

This configuration trusts any artifact in the `com.github.javaparser` group if it's signed with key `8756c4f765c9ac3cb6b85d62379ce192d401ab61`.

The `trusted-key` element supports these attributes:

- `group` - the group of the artifact to trust
- `name` - the name of the artifact to trust
- `version` - the version of the artifact to trust
- `file` - the name of the artifact file to trust
- `regex` - whether to interpret the other attributes as regular expressions (defaults to `false`)
- `origin` - an optional, free-form note documenting where the key was obtained or verified (for example, the URL it was downloaded from)
- `reason` - an optional note explaining why the key is trusted

For example:

```xml
```

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

The `origin` and `reason` attributes are purely informational: Gradle preserves them across read/write cycles but never uses them during verification.

When a single key is trusted for several different groups or artifacts, Gradle groups the entries under one `trusted-key` element with nested `trusting` elements. In that case the key-level `origin` and `reason` act as defaults, and each `trusting` entry may override them with its own values. This is useful when the same key signs very different artifacts whose provenance you want to document separately:

```xml
<trusted-keys>
   <trusted-key id="8756c4f765c9ac3cb6b85d62379ce192d401ab61"
                origin="https://keyserver.ubuntu.com"
                reason="Verified against the maintainer's website">
      <trusting group="com.github.javaparser"/>
      <trusting group="com.example" reason="Trusted internally after manual review"/>
   </trusted-key>
</trusted-keys>
```

Here the `com.github.javaparser` entry inherits the key-level `origin` / `reason`, while the `com.example` entry overrides the `reason`.

Security Considerations

Be careful when trusting keys globally:

- A valid key may have been used to sign artifact `A` which you trust
- Later, the key could be stolen and used to sign malicious artifact `B`

This means you should limit trust to appropriate groups or artifacts, and avoid trusting keys too broadly.

IMPORTANT

Anyone can put an arbitrary name when generating a PGP key. Never trust a key solely based

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

You might want to trust some artifacts more than others. For example, it's legitimate to think that artifacts produced in your company and found in your internal repository are safe, but you want to check every external component.

NOTE

This is a typical company policy. However, nothing prevents your internal repository from being compromised, so it's a good idea to check your internal artifacts too!

You can automatically trust all artifacts in a group by adding this to your configuration:

```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.o
   <configuration>
      <trusted-artifacts>
         <trust group="com.mycompany" reason="We trust mycompany artifacts"/>
      </trusted-artifacts>
   </configuration>
</verification-metadata>
```

This means all components with group `com.mycompany` will be automatically trusted. Trusted means Gradle will not perform any verification whatsoever.

The `trust` element accepts these attributes:

- `group` - the group of the artifact to trust
- `name` - the name of the artifact to trust
- `version` - the version of the artifact to trust
- `file` - the name of the artifact file to trust
- `regex` - whether to interpret the `group`, `name`, `version`, and `file` attributes as regular

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```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.o
   <configuration>
      <trusted-artifacts>
         <trust group="^com[.]mycompany($|([.].*))" regex="true" reason="We trust all mycompany art
      </trusted-artifacts>
   </configuration>
</verification-metadata>
```

Configuring Key Servers, Keyrings, and Keys

Specifying Key Servers

Gradle automatically downloads public keys from well-known key servers. You can configure custom key servers by adding them to your verification metadata:

```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.o
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>true</verify-signatures>
      <key-servers>
         <key-server uri="hkp://my.server.com"/>
         <key-server uri="https://my.key.server.com"/>
      </key-servers>
   </configuration>
</verification-metadata>
```

Using Local Keyrings Only

The local keyring files (`.gpg` or `.keys`) can be used to avoid reaching out to key servers whenever

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

To avoid this, you can disable key servers entirely. Only the local keyring file will be used, and if a key is missing, the build will fail.

To enable this mode, disable key servers in the configuration file:

```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.o
   <configuration>
      <key-servers enabled="false"/>
   </configuration>
</verification-metadata>
```

NOTE

If you are asking Gradle to generate verification metadata and an existing file sets `enabled` to `false`, this flag will be ignored so that potentially missing keys can be downloaded.

Ignoring Unavailable Keys

Sometimes a key is not available because it wasn't published to a public key server or was lost. In these cases, you can ignore the key:

```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.o
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>true</verify-signatures>
      <ignored-keys>
         <ignored-key id="abcdef1234567890" reason="Key is not available in any key server"/>
      </ignored-keys>
   </configuration>
```

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

NOTE

If Gradle cannot download a key while bootstrapping, it will automatically mark it as ignored. If you can find the key but Gradle cannot, you can manually add it to the keyring file.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Maintaining Verification Metadata

Once you've set up dependency verification, you'll need to maintain the verification metadata file and keyring files over time. This includes cleaning up old entries, refreshing keys, and managing your local keyring for better performance.

Cleaning up the Verification File

If you do nothing, the dependency verification metadata will grow over time as you add new dependencies or change versions: Gradle will not automatically remove *unused* entries from this file. The reason is that there's no way for Gradle to know upfront if a dependency will effectively be used during the build or not.

As a consequence, adding dependencies or changing dependency versions can easily lead to more entries in the file while leaving unnecessary entries.

One option to clean up the file is to move the existing `verification-metadata.xml` file to a different location and call Gradle with the `--dry-run` mode: while not perfect (it will not notice dependencies only resolved at configuration time), it generates *a new file* that you can compare with the existing one.

We need to move the existing file because both the bootstrapping mode and the dry-run mode are incremental: they copy information from the existing metadata verification file (in particular, trusted keys).

Refreshing Missing Keys

Gradle caches missing keys for 24 hours, meaning it will not attempt to re-download the missing keys for 24 hours after failing.

If you want to retry immediately, you can run with the `--refresh-keys` CLI flag:

BASH

```
./gradlew build --refresh-keys
```

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

For ASCII-Armored Format (.keys)

If you're using the ASCII-armored keyring format, you can add keys directly by appending them to the `gradle/verification-keyring.keys` file.

First, export the public key to a file:

```
BASH
$ gpg --export --armor 8756C4F765C9AC3CB6B85D62379CE192D401AB61 > key.asc
```

Then append the key to your keyring file:

```
BASH
$ cat key.asc >> gradle/verification-keyring.keys
```

For Binary Format (.gpg)

If you're using the binary keyring format, use GPG to import the key:

```
BASH
$ gpg --no-default-keyring --keyring gradle/verification-keyring.gpg --import key.asc
```

Or import directly from a key server to the keyring:

```
BASH
$ gpg --no-default-keyring --keyring gradle/verification-keyring.gpg --recv-keys 8756C4F765C9AC3CB6
```

NOTE

Remember to use full 40-character fingerprint IDs when working with keys, not short or long IDs.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Gradle supports two keyring formats:

- **Binary format** (`.gpg`) - Compact and can be updated via GPG commands, but not human-readable
- **ASCII-armored format** (`.keys`) - Human-readable, easy to edit manually, and produces readable diffs for code reviews

You can configure which format to use:

```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.o
   <configuration>
      <verify-metadata>true</verify-metadata>
      <verify-signatures>true</verify-signatures>
      <keyring-format>armored</keyring-format>
   </configuration>
</verification-metadata>
```

Available options are `armored` and `binary` .

Without `keyring-format` specified, if `gradle/verification-keyring.gpg` or `gradle/verification-keyring.keys` exists, Gradle will use it. The binary version (`.gpg`) takes precedence if both exist.

Exporting Keys

To export all keys used during verification to the keyring file, run:

```bash
$ ./gradlew --write-verification-metadata pgp,sha256 --export-keys
```

Unless `keyring-format` is specified, this generates both the binary and ASCII-armored files. You

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

NOTE

This command will not report verification errors, only export keys.

Committing Keyring Files

It's a good idea to commit the keyring file to version control (as long as you trust your VCS).

If you use the binary format with Git, add this to your `.gitattributes` file to ensure it's treated as binary:

```
TEXT
*.gpg          binary
```

NOTE

Only public key packets and a single userId per key are stored. All other information (user attributes, signatures, etc.) is stripped from exported keys.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Understanding Signature Verification

Once signature verification is enabled, Gradle follows this workflow for each artifact:

1. Attempts to download the corresponding `.asc` signature file
2. If the signature file exists:
    - Looks for the keys in the keyring, if they are not there, automatically downloads the public keys needed for verification
    - Verifies the artifact using the downloaded keys
    - If signature verification passes, performs any additional checksum verification you configured
3. If the signature file is absent, falls back to checksum verification

This means Gradle's verification is stronger with signatures enabled than with checksums alone:

- If an artifact is signed with multiple keys, all of them must pass validation or the build fails
- If an artifact passes signature verification, any additional checksums configured for that artifact will also be checked

However, passing signature verification doesn't automatically mean you can trust the artifact—you must explicitly trust the keys by adding them to your verification metadata as shown in Updating Verification Metadata Manually.

Bootstrapping with Signatures

When you bootstrap with signature verification enabled using `--write-verification-metadata pgp,sha256`, Gradle takes an *optimistic approach* and assumes signature verification is sufficient. However, you must still provide a fallback checksum algorithm (like `SHA256`) because not all artifacts are signed.

Gradle performs optimistic verification during bootstrapping and automatically

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

- **Auto-generates checksums** - For artifacts without signatures or with ignored keys, Gradle generates checksums using the fallback algorithm
- **Auto-groups keys** - Gradle attempts to group keys at the group level (in `<trusted-keys>`) rather than per-artifact to minimize configuration file size

WARNING

Bootstrapping takes an optimistic point of view that signature verification is enough. If you also care about integrity, you should first bootstrap using checksum verification, then re-bootstrap with signature verification. This ensures you have checksums for all artifacts before adding signatures.

If signature verification fails during bootstrapping, Gradle will automatically generate an ignored key entry but will warn you to investigate. This commonly happens when POM files differ between repositories in non-meaningful ways.

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Troubleshooting Verification Failures

Dependency verification can fail in different ways. This section explains how to deal with various cases.

Missing Verification Metadata

The simplest failure occurs when verification metadata is missing from the dependency verification file. This happens when you update a dependency and new versions (and potentially transitive dependencies) are brought in.

Gradle will tell you what metadata is missing:

```
TEXT
Execution failed for task ':compileJava'.
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact commons-logging-1.2.jar (commons-logging:commons-logging:1.2) in repository 'Mave
```

The missing module group is `commons-logging`, artifact name is `commons-logging`, and version is `1.2`. You need to add an entry to the verification file:

```
XML
<component group="commons-logging" name="commons-logging" version="1.2">
   <artifact name="commons-logging-1.2.jar">
      <sha256 value="daddea1ea0be0f56978ab3006b8ac92834afeefbd9b7e4e6316fca57df0fa636" origin="offi
   </artifact>
</component>
```

Alternatively, ask Gradle to generate the missing information using bootstrapping. Existing information will be preserved—Gradle only adds missing verification metadata.

Incorrect Checksums

A more serious issue is when checksum verification fails:

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Gradle shows the dependency, expected checksum (declared in metadata file), and actual computed checksum.

This indicates a dependency may have been compromised. You must perform manual verification. Several scenarios:

- **Dependency tampered in local cache** - Delete the file from cache and Gradle will re-download.
- **Dependency available in multiple sources with slightly different binaries** - Additional whitespace, timestamps, etc.
  - Inform library maintainers about the issue
  - Use `also-trust` to accept additional checksums
- **Dependency was compromised**
  - Immediately inform library maintainers
  - Notify repository maintainers

Note: A variation is *name squatting* (GAV coordinates that look legit but differ by one character) or *repository shadowing* (official GAV coordinates published in a malicious repository that comes first in your build).

Untrusted Signatures

With signature verification enabled, Gradle verifies signatures but doesn't automatically trust them:

```
TEXT
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-core:3.6.11) in repo
```

You need to verify if the key can be trusted, then refer to Understanding Signature Verification to

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Gradle will fail with:

```
TEXT
> Dependency verification failed for configuration ':compileClasspath':
    - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-core:3.6.11) in repo
```

Options:

1. Signature was wrong in the first place (happens frequently with dependencies published on different repositories)
2. Signature is correct but artifact has been compromised (local cache or remotely)

Go to the official site to verify if they publish signatures. If they do, verify that Gradle's downloaded signature matches the published one.

If you've verified the dependency is not compromised and only the signature is wrong, declare an artifact-level key exclusion and provide a checksum.

Handling Wrong Signatures

There are several options when you encounter a signature verification failure:

1. The signature was wrong in the first place, which happens frequently with dependencies published on different repositories.
2. The signature is correct but the artifact has been compromised (either in the local dependency cache or remotely).

The right approach here is to go to the official site of the dependency and see if they publish signatures for their artifacts. If they do, verify that the signature that Gradle downloaded matches the one published.

NOTE

Artifacts are often signed with expired keys, which is not a problem for verification. Key

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

```xml
<components>
    <component group="com.github.javaparser" name="javaparser-core" version="3.6.11">
       <artifact name="javaparser-core-3.6.11.pom">
          <ignored-keys>
             <ignored-key id="379ce192d401ab61" reason="internal repo has corrupted POM"/>
          </ignored-keys>
       </artifact>
    </component>
</components>
```

However, if you only do so, Gradle will still fail because all keys for this artifact will be ignored and you didn't provide a checksum:

```xml
<components>
    <component group="com.github.javaparser" name="javaparser-core" version="3.6.11">
       <artifact name="javaparser-core-3.6.11.pom">
          <ignored-keys>
             <ignored-key id="379ce192d401ab61" reason="internal repo has corrupted POM"/>
          </ignored-keys>
          <sha256 value="a2023504cfd611332177f96358b6f6db26e43d96e8ef4cff59b0f5a2bee3c1e1"/>
       </artifact>
    </component>
</components>
```

Gradle User Manual
Version 9.8.2
Getting Started

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Gradle's Feature Lifecycle

Gradle Fundamentals

▶Learning Gradle Basics
▶Writing Build Scripts
▶Creating Plugins

Gradle Tutorials

▶Beginner Tutorial
▶Intermediate Tutorial
▶Advanced Tutorial

Gradle Best Practices

Introduction
▶Best Practices

Gradle DSLs and APIs

▶Groovy
▶Kotlin
▶General

Gradle Reference

▶Runtime and Configuration
▶Core Plugins
▶Tasks
▶Plugins
▶Dependencies
▶Gradle Managed Types
▶Platforms
▶Other Topics

Gradle on CI

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Composite Builds
Configuration on Demand

Optimizing Gradle Builds

Improving Performance
▶Build Cache
▶Configuration Cache
Isolated Projects

Securing Gradle Builds

Supply Chain Security
Verifying Dependencies

IDE & Tool Integration

Third-party Tools
▶APIs

How-To-Guides

▶Structuring Builds
▶Dependency Management

More Resources

Licenses
Single Page Version

Stay UP-TO-DATE on new features and news:

By entering your email, you agree to our Terms (https://develocity.ai/legal/terms-of-service/?utm_source=docs.gradle.org&utm_medium=referral&utm_content=%2Fcurrent%2Fuserguide%2Fdependency_verification.html) and Privacy Policy (https://develocity.ai/legal/privacy/?utm_source=docs.gradle.org&utm_medium=referral&utm_content=%2Fcurrent%2Fuserguide%2Fdependency_verification.html)

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking “Accept,” below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.

Email address Subscribe

© 2026 Gradle, Inc. Gradle®, Develocity®, Build Scan®, and the Gradlephant logo are registered trademarks of Gradle, Inc.

(https://gradle.org)

Privacy (https://develocity.ai/legal/privacy/?utm_source=docs.gradle.org&utm_medium=referral&utm_content=%2Fcurrent%2Fuserguide% | Terms of Service (https://develocity.ai/legal/terms-of-service/?utm_source=docs.gradle.org&utm_medium=referral&utm_content=%2Fcurrent%2Fuserguide%

We'd like to collect non-essential cookies for analytics and marketing which involves cookies managed by third parties. You can read more about how we use cookies in our Privacy Policy (https://develocity.ai/legal/privacy//) . By clicking "Accept," below you agree to our website's cookie use as described in our Privacy Policy and our collection of non-essential cookies.