Enhancing Validation Through Attestations

Brent Clausner

beclausner@sei.cmu.edu

Abstract

Validation of software involves a variety of tests and processes to make it a robust product. Each step is essential to build a release. Utilizing attestations provide proof that the process was followed and that no unauthorized changes have been made throughout creating build artifacts. These attestations can be provided along with the software release package. Providing attestations and a plan layout to your customer gives them confidence that your organization has produced what is provided. Several checks and balances are included. This can eliminate the manual process of performing checksum examinations. This isn't an industry requirement yet and the responsibility lies with the person installing the software. An open-source project, like in-toto, can help to make the supply-chain of your product more secure. These steps can be added to your build pipeline to streamline the whole process. This paper will demonstrate how attestations can enforce policy and give consumers confidence that they have correct products.

Biography

Brent Clausner is a DevOps Engineer at The Software Engineering Institute. His background is rooted in software development having worked as a Software Engineer, Quality Assurance Engineer, and DevOps Engineer. He is a proponent to have projects adhere to the best practices and advocates for robust secure code being implemented.

1 Introduction

Typically, when someone purchases or licenses software for use, they download the release package, any binary executables or libraries, and should check the checksum hash against what is provided by the company you're purchasing or licensing from. This also includes any open-source software. This is meant to help ensure that you are getting what is expected. Without doing that you risk a man-in-the-middle attack. Including more protection for the customer in this case is always a good thing. If the packaged software includes multiple executables and reference files, doing this can be a daunting task. As an end user of any software, you should have a way to validate that the obtained software followed the providers expected process and you obtained the same released software.

Making use of attestations from your software suppliers can help to prevent attacks like what happened with Solar Winds. Using attestations goes beyond simple checksum verification as it is included in the attestation verification process. Providing attestations and layouts for your software product, helps to prevent man in the middle attacks and allows your customers to be sure they got what you intended.

The purpose of this document is to go over what processes can help make the customer aware that not only your build and testing process was followed, but also protecting internal code and output from commands from leaking to the public. We will be exploring the use of in-toto within an organization as a means of validating all testing steps. Doing this, will give the consumer of your product a way to have more trust in your software.

2 Attestations

A software attestation can be defined as, [1] "An authenticated statement (metadata) about a software artifact or collection of software artifacts". These are small files that contain a lot of metadata.

2.1 Attestation Contents

Attestations can contain a variety of metadata about something in the build process. Often this includes who executed a command, what output the command produced, environment variables, the command return code, what files exist before the command ran, what files exist after a command ran, and all the sha256 checksum for every file.

Figure 1: Envelope and contents

[5] According to Supply chain Levels for Software Artifacts Committee (SLSA), an attestation must contain 3 things.

1. An envelope which contains a message containing the content in a statement and a signature.

2. A statement that bines the attestation to a set of artifacts. That statement is made up of a subject that identifies the artifact.

3. A predicate that contains metadata about the subject which normally contains a link.

2.2 Current Tooling

A few tools exist to allow generation and validation of attestations. This paper will focus on, In-toto. It is an open-source tool that allows you to generate and verify attestations. The commands can wrap whatever existing commands you have to produce those attestation files.

[2] Witness is an open-source tool that allows you to generate attestations, and can be used for verifying those attestations. It implements in-toto specification, ITE-5, ITE-6, and ITE-7. In-toto Enhancements (ITE) are used to track specific functionality for in-toto. These enhancements are specifications similar to a key to an issue tracking for in-toto.

[3] Another tool is a Google based Kubernetes solution called Binary Authorization. It can be used to verify images are signed by trusted authorities. This could be used along with in-toto.

2.2.1 In-toto terminology

This portion is important to describe terms used in this paper that are related to In-toto.

[4] **Step** is referenced as a single action that is signed by the functionary that executed it.

[4] **Inspection** is a client-side validation step. This gets included in a layout that can do additional validation outside of command file create/delete.

[4] **Layout** is a file that specifies each step necessary and who is supposed to execute the commands to create a final product. It also specifies what materials and products are expected at each step. Inspections are also able to be specified here as well to do additional validation.

[4] **Materials** are specified as files that are expected to be there at a step. These can be products from other steps. It consists of a list under the tag “expected_materials”. Files are specified as a string and is a relative path to the file.

[4] **Products** are the files that are expected to be there because of a command. Materials can be included from other steps. It consists of a list under the tag “expected_products”. Files are specified as a string and is a relative path to the file.

2.3 Layouts

The layout has all the steps necessary to create a final product. It’s easiest to use a python file to generate the layout for you. Layouts can expire so it is useful to have it capable to be regenerated by having the layout defined in python. The layout is in JSON format. It being read in to the *“in_toto.models.layout”* module. With python’s use of whitespace this makes it easy to read though it can be very long.

It consists of the “payload” that is a generated string based off the python JSON input and signers’ signature. The string is not human readable as it appears hashed off those two inputs. This helps in preventing bad actors from manipulating it and requires that the generator have private key access.

2.3.1 In-toto Layout Steps

The contents of the layout consist first of steps. The “steps” tag is a list that consists of several dictionaries. The steps define what files are allowed to exist in the “expected_materials” tag. The files are specified as a string with a full relative path. Files listed outside of this JSON list within the layout will cause the verification to fail. Wildcards are acceptable if you have a source code directory that has many files to be made and removed. Similarly, with products of that step, you can specify the directory for build objects with a wild card for creation. This is done by using the “expected_products” tag to list what files will exist and should exist after the command of that step is run. Files being created that are not in that list will cause the verification command to fail.

The command that the step is, is a list under the tag of “expected_command”. For instance, if you were to want to build a rust project using cargo, normally the command is “cargo build”. With it being a list it is argument separated so it would be “[‘cargo’, ‘build’]”.

```
{
...
"steps": [
 { "name": "...",
   "expected_materials": [ ... ],
   "expected_products": [ ... ],
   "expected_command": [ ... ],
   "pubkeys": [ ... ],
 },
 ...]
}
```

Figure 2 JSON Sample of Steps in a Layout

Both materials and products have similar syntax. They consist of a list and start with a keyword. The keyword determines what action has taken place to determine if a file is valid or not. Those keywords are:

- "ALLOW" is meant to indicate that a file is allowed to be there.
- "CREATE" is meant to indicate that a file is expected to be created because of the command. This is an "expected_product" specific keyword.
- "MATCH" keyword allows you to match materials or products from other steps, even within the same step. For example, a product can match the materials from the same step.
- "DISALLOW" keyword is normally the last entry in the list. Typically is followed by the "*" to have it error on any unexpected files that are created or are already there that aren't intended to.

2.3.2 Private and public keys

Currently a few methods are supported, they include RSASSA-PSS, ED25519, and ECDSA. The private key is used for either signing the layout and/or signing the creation of the attestation. I recommend having a separate signature that's used for signing the layout creation and use the public keys of user accounts that will execute the steps. Keeping the keys secure is the most important factor to using this tool. With the layout signer private key, a bad actor would have the ability to change what commands are run as part of the layout. Private keys must be guarded.

2.4 Attestation generation

In general, the command gets wrapped by "in-toto-run" command. Several things get passed into the wrapping command, such as the private key, name of the step, timeout, and finally the command to execute. The result will be that a file referenced as a link file will be created. This file will be named as step name, followed by a keyed prefix, and finally the extension of "link". That "link" file is the attestation.

3 Intended Attestation Usage

The normal way an attestation is intended to be generated is part of the build process. Normally the build process consists of cloning of a repository, build/create the executable, and package the build artifacts.

This process is to ensure that the end build executable is built from the correct source in the manner laid out by the layout. This is referred to what is listed under the “expected_materials” tag. The result is all of the software’s package materials, attestations, and layout would be provided to the consumer of the software so they can verify it themselves. This provides evidence that the build/files all match that were used throughout the entire build. Nothing was modified when it wasn’t expected to.

With all the attestations and the build software, executing the command “in-toto-verify”, along with its command line arguments, “in-toto-verify” will verify all the attestation file contents with each step. This replaces checking the shasum against whatever is stated at the source of download. This method goes further into verifying all components in the entire process.

In-toto-verify will also run “inspection” commands that exist in the layout. The “inspection” is a list of commands that run and specifies what to check. For example, if your software compresses with tar, an inspection command could be to uncompress the file with untar. The “expected_materials” for that inspection should contain the tar file matched to a “step” within the layout. The ”expected_products” tag should include the files matched with other steps in that layout for the tarred file.

4 Proposed Attestation Usage

4.1 Internal Company Usage

Using attestations can be done with a pipeline to ensure that the build comes from where it’s expected to and built in the way dictated by the company that is producing the software. These can be packaged up and pushed to an artifact repository. The artifacts can be downloaded and verified that they match the build through the layout for that company.

When you execute commands there can be a lot of information in that command. The arguments to a build command might describe paths for files or even some sensitive information like passwords. Output from those commands can also show parts of source code, documentation, or again sensitive information like passwords. This information, if it is made public, can have a significant negative impact to a company.

What I’m proposing is that anyone that is doing testing on software that is through their build system, to verify the build artifacts with attestations and a layout as a first step. This attestation won’t be made public as only the verification command will be logged. From here any number of layouts can be used. As some manual testing inevitably will be done on the build, I propose to use a command that would be something like a command that doesn’t do anything, like “echo PASS” to indicate that testing has passed. That command will just output the string “PASS”. If you have multiple teams, a final layout can be used that contains checks for an attestation created by those teams and signed by a team lead or program manager.

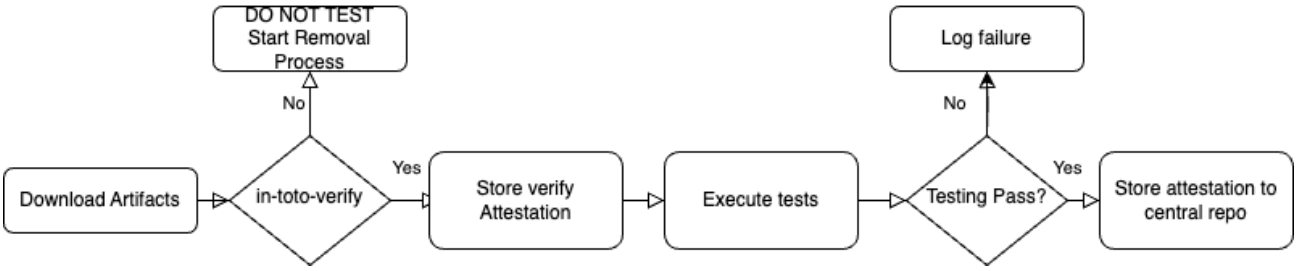

Figure 3 Proposed flowchart of actions

With multiple teams the flowchart above can be followed. You should be storing of all the attestations onto a central server or repository when all tests pass. The only attestations that will be provided to any

consumers will be the ones that issue “in-toto-verify”. This will prevent giving internal build output to end users, though they can verify that the build has been carried through the rigors of the testing process and has been delivered to the end user by the verification of the attestations with “in-toto-verify”.

This also provides the company a trail of who did what to ensure quality in the software product. These attestations give proof that someone is signing a file that testing has been done.

4.1.1 Hypothetical outcome

If you are working for a company that for instance would create an operating system, your organization is most likely going to have a lot of different functional teams. If something happens where a function of that operating system does not work as expected when saving files to disk and that bug is found after the product is released by a customer, this can give the company visibility into where the gap is.

With attestations, it allows the user to verify the exact version is from the expected manufacturer. It can help to prevent man in the middle attacks. This also allows the end user the capability to provide the manufacturer those attestations when there is a problem such as a claim above. They will most likely request other additional information such as log files. The exact version could be determined from that content and that it is in fact the manufacturers product.

A claim that is found without this type of protection can lead to a false line of validation. Without having a manner to determine the exact version of software that a user is using and that it hasn’t been tampered with, this wouldn’t be worth investigating. This could happen if it was downloaded from a third party website. If there are not attestations, layout, and public keys, it makes it difficult to truly verify that the software is from the expected manufacturer.

4.2 Quality Assurance Testing

Using multiple layouts from within a company can help to ensure the necessary process is followed. For instance, including known processes either via a pipeline or manually of doing static code analysis is a good thing to do. The output and command arguments may not be something that you would want consumers to have access to.

There may be things in code analysis, such as a static code analysis “lint” check, that you are ok with doing that tools pick up as a violation. Something like white spacing can often show up. When that does it can include lines of code in the output. The output of the command is in inside the attestation and that could lead to leaking the information outside the organization if are shared.

4.2.1 Multiple Layouts

Using a layout that is from a build process to verify the version and executable is necessary as a first step for anyone doing testing. The verification step of the build should have an attestation created for that. That should be the first layout that would be in use.

The following JSON is an example for the first layout. You can see where files could be, like under “src/*” and what commands are used to build, “cargo build”.

```
layout = Layout.read({
    "_type": "layout",
    "keys": {
        key_owner["keyid"]: key_owner,
    },
    "steps": [
        {
```

```
            "name": "building",
            "expected_materials": [
                ["ALLOW", "src/*"], ["ALLOW", "*.link"], ["ALLOW", "*.pub"],
                ["ALLOW", "*.layout"], ["ALLOW", "Cargo.toml"]
            ],
            "expected_products": [
                ["MATCH", "*", "WITH", "MATERIALS", "FROM", "building"],
                ["CREATE", "target/*"], ["CREATE", "Cargo.lock"],
                ["DISALLOW", "*"]
            ],
            "expected_command": [
                "cargo", "build"
            ],
            "threshold": 1,
            "pubkeys": [key_owner["keyid"]],
        },
        {
            "name": "lint_check",
            "expected_materials": [
                ["MATCH", "*", "WITH", "MATERIALS", "FROM", "building"],
                ["MATCH", "*", "WITH", "PRODUCTS", "FROM", "building"],
                ["DISALLOW", "*"]
            ],
            "expected_products": [
                ["MATCH", "*", "WITH", "MATERIALS", "FROM", "building"],
                ["MATCH", "*", "WITH", "PRODUCTS", "FROM", "building"],
                ["DISALLOW", "*"]
            ],
            "expected_command": [
                "cargo", "clippy"
            ],
            "threshold": 1,
            "pubkeys": [key_owner["keyid"]],
        },
        {
            "name": "package",
            "expected_materials": [
                ["MATCH", "*", "WITH", "MATERIALS", "FROM", "building"],
                ["MATCH", "*", "WITH", "PRODUCTS", "FROM", "building"],
                ["DISALLOW", "*"]
            ],
            "expected_products": [
                ["MATCH", "*", "WITH", "MATERIALS", "FROM", "building"],
                ["MATCH", "*", "WITH", "PRODUCTS", "FROM", "lint_check"],
                ["CREATE", "package.tar"],
                ["DISALLOW", "*"]
            ],
            "expected_command": [
                "tar", "-cf", "package.tar", "target/debug/simple",
            ],
            "threshold": 1,
            "pubkeys": [key_owner["keyid"]],
        },
    ],
```

```
    "inspect": [],
})
```

The layout above is used with creating and verifying the build. The verification of that layout and attestations should be the first step in doing manual testing. The ‘in-toto-verify’ should be wrapped in the ‘in-toto-run’ command to generate the attestation. After all the manual testing is completed, if all tests passed, executing the command “echo PASS” wrapped by the ‘in-toto-run’ command. This will generate an attestation that can be provided to a release manager, who can run the last verification command.

Having a layout for a step prior to release should include verifying that each team did their own verification of the build components, executed tests with an attestation from a designated lead for that team, and updated test tracking software. This would be a second layout that can be used. At this point, the layout, attestations, and public keys could be provided to the public for verification, but that can be a lot of files and additional space being required wherever the release files are stored.

```
    layout = Layout.read({
        "_type": "layout",
        "keys": {key_owner["keyid"]: key_owner},
        "steps": [
            {
                "name": "verify",
                "expected_materials": [
            ["ALLOW", "in-toto-files.tar"], ["ALLOW", "*.link"], ["ALLOW", "*.pub"],
            ["ALLOW", "package.tar"], ["ALLOW", "*.layout"],
            ["ALLOW", "target/debug/simple"], ["DISALLOW", "*"]
                ],
                "expected_products": [
                    ["MATCH", "*", "WITH", "MATERIALS", "FROM", "verify"],
                    ["DISALLOW", "*"]
                ],
                "expected_command": [
                    "in-toto-verify", "-l", "./build.layout",
                    "--verification-keys", "./owner.pub", "--link-dir", "./"],
                "threshold": 1,
                "pubkeys": [key_owner["keyid"]]},
            {
                "name": "testing",
                "expected_materials": [
                    ["MATCH", "*", "WITH", "MATERIALS", "FROM", "verify"],
                    ["DISALLOW", "*"]],
                "expected_products": [
                    ["MATCH", "*", "WITH", "MATERIALS", "FROM", "verify"],
                    ["DISALLOW", "*"]],
                "expected_command": [
                    "echo", "PASS"
                ],
                "threshold": 1,
```

```
                "pubkeys": [key_owner["keyid"]],
            }
        ],
        "inspect": []
    })
```

The above layout will validate that the build is what's expected and can be expanded for multiple teams with the step "testing" above. Additional steps can be added there that would signify the functional teams and system testing teams that an organization might be using. The verification of this should be used as the last attestation that would be public facing. The output from this command will only show any information from the 'in-toto-verify' command. This will not include any output of actual code from the build process.

Example layout that can be used for release:

```
layout = Layout.read({
    "_type": "layout",
    "keys": {key_owner["keyid"]: key_owner},
    "steps": [
        {
            "name": "release",
            "expected_materials": [
                ["ALLOW", "package.tar"], ["ALLOW", "*.link"],
                ["ALLOW", "*.pub"],
                ["ALLOW", "*.layout"], ["ALLOW", "target/debug/simple"],
                ["DISALLOW", "*"]
            ],
            "expected_products": [
                ["MATCH", "*", "WITH", "MATERIALS", "FROM", "verify2"],
                ["ALLOW", "*.layout"], ["ALLOW", "target/debug/simple"],
                ["DISALLOW", "*"]
            ],
            "expected_command": [
                "in-toto-verify", "-l", "./internal.layout",
                "--verification-keys", "./owner.pub", "--link-dir", "./"
            ],
            "threshold": 1,
            "pubkeys": [key_owner["keyid"]],
        }
    ],
    "inspect": []
})
```

Having a simpler layout as a final version for release, as shown above, should include the verify step to check the layout, before release. The output for the attestation produced for public release would only include the command, under "expected_command", to verify the testing layouts. The included file metadata from this would be the files that are for release have matched to the ones in all the layouts previous.

The examples shown above is an extremely simplified version of what I'm proposing. Additional test steps could be added to the first layout for quality assurance pieces or additional layouts could be added, with

multiple verify steps within second layout before the release layout. There are a lot of ways to handle this. The sample above using three layouts does work. The second layout that includes a verify of the "build.layout" of the first would be a good example of what any test engineer could run prior to capturing tests. The release manager should still have a small layout at the end that has a layout that includes all the test attestation and verifies it. That verify attestation would be what's provided to a customer.

4.3 Customer Deliverables

The following should be provided to customers:

- Final verify attestation
- Release layout
- Public keys
- Release software

If the release attestation is generated in a folder that only includes the links and release materials, the link files that were used for the verify step within will be excluded from what the consumer will receive. This means the filenames and output from commands will not be shipped to them, just the materials necessary to run the software and the files for verification.

5 Conclusion

There is an initial up front cost for setting this up. Having someone to coordinate the layouts and maintain them is a cost. Once they are created, maintaining them isn't as difficult as they can be modified and regenerated on demand. The software release manager should oversee the final layout. They will be required to work with leads from functional teams to coordinate the attestations from them and the expected commands and layouts in use for the various teams.

The final layout that would be provided to customers should be small and include a single "in-toto-verify" command. This reduces the output captured for the attestation and ensures that the entire build/release chain has been validated the whole way through.

5.1 Customer Impact

Having all these together allow a customer to verify everything occurred from the perspective of the manufacturer. If there ends up being major complications for what the software was meant to do, it can be traced all the way back to the test team. As long as the manufacturer has stored testing logs, it can be seen if there were any mistakes made in the testing process. This holds the manufacturer accountable for what they're doing.

This should give customers more trust in the manufacturer. One thing to know about downloading software is that most companies provide some hash shasum that can be used to verify the downloaded object is the expected version. Using the in-toto verification adds to this process and checks all of those for you, so it is very straightforward for a customer to verify the release software.

References

[1] SLSA Committee, “SLSA * Terminology”, Supply chain Levels for Software Artifacts, https://slsa.dev/spec/v1.0/terminology (accessed May 6, 2024)

[2] The Witness Contributors, “Witness | Witness In-toto”, Witness.dev, https://witness.dev/docs/ (accessed May 6, 2024)

[3] “Binary Authorization | Google Cloud”, Google Inc., https://cloud.google.com/binary-authorization (accessed May 6, 2024)

[4] ”in-toto-spec.md”, In-toto, https://github.com/in-toto/docs/blob/v1.0/in-toto-spec.md#17-terminology (accessed May 6, 2024)

[5] SLSA Committee, “SLSA * Attestations”, Supply chain Levels for Software Artifacts, https://slsa.dev/attestation-model (accessed May 8, 2024)

Figure 1: Licensed from Adobe Stock images and modified for this paper.

Figure 2: Produced by Brent Clausner

Figure 3: Produced by Brent Clausner