

in-toto Specification

Apr 11, 2017

<https://in-toto.io>

Version 0.8

1. Introduction

1.1. Scope

This document describes in-toto, a system for securing the way in which software is developed, built, tested, and packaged (i.e., the software supply chain). in-toto attests to the integrity and verifiability of all the actions performed while writing code, compiling, testing, and deploying software. It does so by making it transparent to the user what steps were performed, by whom and in what order. As a result, given guidance by the group creating the software, in-toto allows the user to verify if a step in the supply chain was intended to be performed, if the step was performed by the right actor, and attests that materials (e.g., source code) were not tampered with between steps.

1.2. Motivation

Supply chain security is crucial to the overall security of a software product. An attacker who is able to control a step in the supply chain can alter the product for malicious intents that range from introducing backdoors in the source code to including vulnerable libraries in the final product. Because of its susceptibility to these threats, a supply chain breach is an impactful means for an attacker to affect multiple users at once.

Although many frameworks ensuring security in the "last mile" (e.g., software updaters) exist, they may be providing integrity and authentication to a product that is already vulnerable; it is possible that, by the time the package makes it to a software update repository, it has already been compromised.

1.3. History and credit

September 19, 2016 -- Version 0.1 of this document is released

1.4. Context

A software supply chain is the series of steps performed when writing, testing, packaging, and distributing software. In a typical software supply chain, these multiple steps that transform (e.g., compiling) or verify the state (e.g., linting) of the project are "chained" together in order to drive it to a final product.

1.5. Goals

in-toto aims to provide integrity, authentication and auditability to the supply chain as a whole. This means that all the steps within the supply chain are clearly laid out, that the parties involved in carrying out a step are explicitly stated, and that each step carried out meets the requirements specified by the actor responsible for this software product.

1.5.1 Goals for implementation

- The client side of the framework must be straightforward to implement in any programming language and for any platform with the requisite cryptographic support (as described in sections 4.1 and 4.2).
- The framework should be easy to integrate with existing installers, build systems, testing frameworks, version control systems, and software update systems.
- The framework should be easily customizable for use with any cryptographic library.
- The process by which functionaries gather metadata about the steps carried out must be simple and nonintrusive. This should also be true for the process by which project owners define the supply chain.

1.5.2 Defender goals and non-goals

In the context of in-toto, a defender is the client (i.e., the person who will install the software product). We assume the actors performing steps in the supply chain (functionaries) and the actors in charge of a software product (project owners) are not acting maliciously. With this in mind, we describe the following goals for a defender.

Goals:

The user's goal is to install a target software product that has gone through all the steps of the supply chain and is identified as legitimate. This means that no step was changed, removed, or added into the software development process that the project owner intended. To verify legitimacy, the user also wants to make sure that the software installed came from the designated project owner.

Non-goals:

We are not trying to protect against a functionary (or actor in the supply chain) who unwittingly introduces a vulnerability while following all steps of the supply chain as designed. We are trying to prevent such functionaries from performing operations other than the ones intended for a supply chain. For example, if two people are in charge of running the packaging scripts, in-toto can verify that this is the case by verifying in-toto metadata regarding this operation. We assume that there will not be two colluding (or deceived) developers who jointly introduce a vulnerability.

We also assume the project owner laid out the supply chain (using a supply chain layout as described in section 4.3) so that testing, code review, and verification into the software supply chain gives meaningful security and quality guarantees. While we provide auditability properties so third parties can study and assess the steps of the supply chain to ensure that defenders follow best practices regarding software quality, we do not enforce any specific set of rules. For example, it is possible to configure the supply chain layout so that no code review is performed and a package is built on an untrusted server --- which is an incredibly insecure configuration. While this may be visible to users installing the software, in-toto's role is not to judge or block layouts that are insecure. However, tools that integrate into in-toto may independently block or make judgments about the security of a specific layout.

1.5.3 Assumptions

The client side tools should perform key-management on behalf of the user. Apart from the layout key, the client is never required to retrieve and provide keys for verification. We are exploring mechanisms to distribute layout keys securely.

For cases where trust delegation is meaningful, a functionary should be able to delegate full or limited trust to other functionaries to perform steps on her or his behalf.

1.5.4 System properties

To achieve the goals stated above, we anticipate a system with the following properties:

- **Final product authentication and integrity:** the product received by the client was created by the intended party. This ensures the final product matches bit-by-bit the final product reported by the last step in the supply chain.
- **Process compliance and auditability:** the product received by the client followed the layout specified by the project owner. All steps described have their materials and products correctly linked together, and, if audited by a third party, he or she can verify that all steps were performed as described. For example, within the in-toto metadata, it is possible to see the unit test server's signed statement that the software passed all of its unit tests, or check git commit signatures to validate that a certain code review policy was used.
- **Traceability and attestation:** the conditions under which each step within the supply chain was performed can be identified, as well as the materials used and the resulting products.
- **Step authentication:** the actor who carries out different steps within the supply chain provides evidence of the step using an unforgeable identifier. This means if Alice tagged a release, the evidence provided could only be produced by Alice.
- **Task and privilege separation:** the different steps within the supply chain can be assigned to different parties to perform. This means, if Alice is the only functionary allowed to tag a release, releases tagged by Bob will not be trusted if present in the supply chain.

In addition, the framework must provide an interface to interact with client systems that further verify the integrity of each step.

1.6. Terminology

- **Software supply chain** (or SSC): the series of actions performed to create a software product. These steps usually begin with users committing to a version control system and end with the software product's installation on a client's system.
- **Supply chain layout** (or simply layout, or layout metadata): a signed file that dictates the series of steps that need to be carried out in the SCC to create a final product. The layout includes ordered steps, requirements for such steps, and the list of actors (or functionaries) in charge of carrying out every step. The steps within the supply chain are laid out by a project owner.
- **Project owner**: the authoritative figure within a project. The project owner will dictate which steps are to be carried out in the supply chain, and who is authorized to carry out each step (i.e., define the layout).
- **Functionary**: an individual or automated script that will perform an action within the supply chain. For example, the actor in charge of compiling a project's source code is a functionary.
- **Supply chain step**: a single action in the software supply chain, which is performed by a functionary.
- **Link**: metadata information gathered while performing a supply chain step, signed by the functionary that performed it. This metadata includes information such as materials, products and byproducts.
- **Materials**: the elements used (e.g., files) to perform a step in the supply chain. Files generated by one step (e.g., .o files) can be materials for a step further down the chain (e.g., linking).
- **Product**: the result of carrying out a step. Products are usually persistent (e.g., files), and are often meant to be used as materials on subsequent steps. Products are recorded as part of link metadata.
- **Byproducts**: indirect results of carrying out a step, often used to verify that a step was performed correctly. A byproduct is information that will not be used as a material in a subsequent step, but may provide insight about the process. For example, the stdout, stdin and return values are common byproducts that can be inspected to verify the correctness of a step. Byproducts are recorded as part of link metadata.
- **Final product**: the bundle containing all the files required for the software's installation on the client's system. This includes link metadata, layout metadata, target files, and any additional metadata required for the product's verification (e.g., a digital signature over the installation files).
- **Client inspection step**: a step carried out on the client's machine to verify information contained in the final product.
- **Verification**: the process by which data and metadata included in the final product is used to ensure its correctness. Verification is performed by the client by checking the supply chain layout and links for correctness, as well as by performing the inspection steps.
- **Target files**: Any file that is not part of in-toto metadata (i.e., not a layout or a link file). In the case of an installer, these files will often be unpacked from within the final product and made ready for use on the user's system.

2. System overview

The main goal of in-toto is to provide authentication, integrity and auditability guarantees for the supply chain that created a final product that a client will install.

To avoid ambiguity, we will refer to any files in the final product that in-toto does not use to verify supply chain integrity as "target files." Target files are opaque to the framework. Whether target files are packages containing multiple files, single text files, or executable binaries is irrelevant to in-toto.

The portion of the in-toto layout describing target files is the information necessary to indicate which functionaries are trusted to modify or create such a file. Additional metadata, besides layout and link metadata, can be provided along with target files to verify other properties of the supply chain (e.g., was a code review policy applied?) when inspecting the final product.

The following are the high-level steps for using the framework, as seen from the viewpoint of an operating system's package manager. This is an error-free case:

1. The **project owner** creates a **layout**. This describes the steps that every functionary must perform, as well as the specific inspection steps that must be performed on the client's machine.
2. Each **functionary** performs his or her usual tasks within the supply chain (e.g., the functionary in charge of compilation compiles the binary), and records **link** metadata about that action. After all steps are performed by functionaries, the metadata and target files are aggregated into a final product.
3. The **client** obtains the final product, and **verifies** that all steps were performed correctly. This is done by checking that all materials used were products of the intended steps, that each step was performed by the authorized functionary, and that the layout was created by the right project owner. If additional verification is required on the accompanying metadata (e.g., to verify VCS-specific metadata), the client will then perform additional inspection steps. If verification is successful, installation is carried out as usual.

2.1. Involved parties and their roles

In the context of in-toto, a *role* is a set of duties and actions that an actor must perform.

In the description of the roles that follows, it is important to remember that the framework has been designed to allow a large amount of flexibility for many different use cases. Given that every project uses a very specific set of tools and practices, this is a necessary requirement for in-toto.

There are three roles in the framework:

- **Project owner:** defines the layout of a software supply chain
- **Functionary:** performs a step in the supply chain and provides a piece of link metadata as a record that such a step was carried out.

- **Client:** Performs verification on the final product by checking the provided layout and link metadata.

In addition, there are third-party equivalents of the above roles, which are managed by the delegation mechanism, described in section 2.1.3. We will elaborate on these roles in depth now.

2.1.1 Project owner

As previously stated, the project owner sets the required steps to be performed in the supply chain. For each step, its requirements, and the specific public keys that can sign for evidence of the step are included to ensure compliance and accountability. In addition, the layout file will contain the definition of inspection steps to be carried out when verifying the final product.

2.1.2 Functionaries

Functionaries are intended to carry out steps within the supply chain, and to provide evidence of this by means of link metadata.

A functionary is uniquely identified by the public key that he or she will use to sign a piece of link metadata as evidence that a step within the supply chain was performed.

A functionary can delegate a step of the supply chain to one or more third-parties by creating a *delegated layout*. In this case, a subset of the steps to be performed are defined by such a functionary, who adopts the role of a delegated project owner.

2.1.3 Clients

Clients are users or automated tools who want to use the product.

The client will perform verification on the final product. This includes verifying the layout metadata, and that the link metadata provided matches the specified layout described in the metadata, and performing inspection steps to ensure that any additional metadata and target files meet the criteria specified by the layout for this inspection step.

A client will likely not interact with the in-toto framework directly, as it should be integrated into system installation tools, or package managers.

2.1.4 Delegating steps to a third-party

Delegations allow a functionary to further define steps within the supply chain. When a functionary performs a delegation, instead of designating the next step, she or he will define the series of steps required as if she or he was a delegated project owner. This is helpful if the project owner does not know the specifics of a step, but trusts a functionary to specify them later.

Delegations can also be used for third-party sections of the supply chain. For example, a package maintainer for a Linux distribution will likely delegate all the steps in the version control system to the upstream developers of each package.

2.2 in-toto components

A in-toto implementation contains three main components:

- A tool to generate and design supply chain [layouts](#). This tool will be used by the project owner to generate a desired supply chain layout file. There are many tools in the reference implementation to aid project owners in creating and signing layouts.
- A tool that functionaries can use to create [link metadata](#) about a step. For example, in the reference implementation, this tool is called "in-toto-run.py".
- A tool to be used by the client to perform verification on the final product. This tool uses all of the link and layout metadata generated by the previous tools. It also performs the inspection steps, as directed by the layout. This tool is often included in a package manager or system installer. In the case of the reference implementation, the tool performing this operation is "in-toto-verify.py".

2.3 System workflow example

To exemplify how these roles interact, we will describe a simple scenario. We provide more specific scenarios in section 5.2, after we have presented a more thorough description of the framework.

Consider a project owner, Alice, and her two functionaries, Diana and Bob. Alice wants Diana to write a Python script (foo.py). Then, Alice wants Bob to package the script into a tarball (foo.tar.gz). This tarball will be sent to the client, Carl, as part of the final product. Carl's target file is foo.tar.gz.

When providing the tarball to Carl, Alice will create a layout file that Carl will use to make sure of the following:

- That the script was written by Diana
- That the packaging was done by Bob
- That the script contained in the tarball matches the one that Diana wrote, so if Bob is malicious or if the packaging program has an error, Carl will detect it.

In order to do this, Carl will require four files in the final product: first, Alice's layout, describing the requirements listed above. Then, a piece of link metadata that corresponds to Diana's action of writing a script, and a piece of link metadata for Bob's step of packaging the script. Finally, the target file (foo.tar.gz) must also be contained in the final product.

When Carl verifies the final product, his installer will perform the following checks:

1. The layout file exists and is signed with a trusted key (in this case, Alice's).
2. Every step in the layout has one or more corresponding link metadata files signed by the intended functionaries, as described in the layout (in this case, the link metadata provided by Bob and Diana).
3. All the materials and products listed in the link metadata match, as specified by the layout. This will be used to prevent packages from being altered without a record (missing link metadata), or tampered with while in transit. In this case, the products reported by Diana should match the materials reported by Bob and so on.

4. Finally, as is specified in the layout metadata, inspection steps are run on the client side. In this case, the tarball will be inspected to verify that the extracted `foo.py` matches the one that was written by Diana.

If all of these verifications pass, then installation continues as usual.

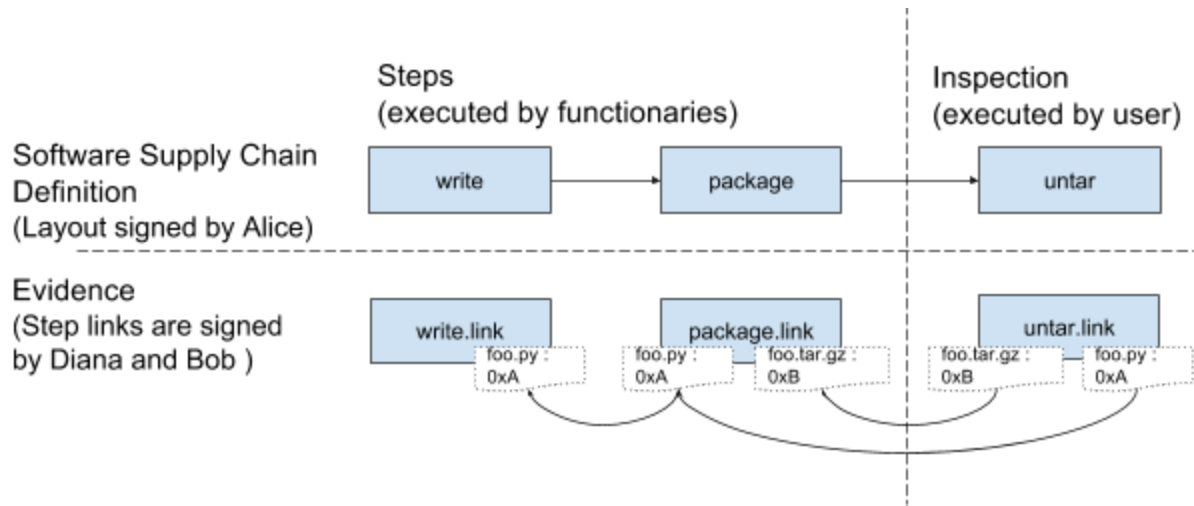


Figure 1: The supply chain pieces for this example

3. The final product

The final product is the bundle of link metadata, layout metadata and target files that will be used by the client's system. Additional, project-specific metadata can be also bundled in the final product for verification during inspection steps.

An installer or package manager uses the framework to inspect the final product and verify its contents. Each project will have specific requirements to verify. For example, a project may want to impose a review policy on the VCS. Thus, it requires in-toto to validate additional accompanying link and layout metadata to verify the review policy was followed.

3.1 Contents

The final product must contain at least these three files:

- The supply chain layout
- A link metadata file
- A target file

More complex and robust supply chain layouts will contain more pieces of link metadata, as well as additional layout files for delegated layouts. Additional metadata (e.g., a signed git commit log) can also be provided to be used during inspection phases.

3.1.1 Supply chain Layout

The supply chain layout specifies each of the different steps and its requirements, as well as the public keys used by functionaries to sign the link metadata for steps within the chain.

The layout will also specify how each piece of link metadata will be verified, and how the chain steps are interconnected via their materials and products.

3.1.2 Link metadata

Link metadata is a statement that a step was carried out. Each piece of link metadata will be used by the framework to ensure that materials and products have not been altered in an unauthorized manner (e.g., while in transit), and, that any alterations have been done only by an intended functionary.

3.1.3 Target files

Target files are the files clients will install and use in their systems. For example, a target file could be an installation disk image, which will be bundled with link metadata for each step performed to create the target file.

3.1.4 Additional metadata files

Additional metadata files can be shipped within the final product for verification. In this case, inspection steps that utilize this metadata can be declared to determine if this metadata is correct. These metadata files will be treated as regular target files by the framework.

4. Document formats

In order to achieve the properties described in Section 1, link and layout metadata must contain information that correctly depicts which operations are intended, and that presents specifics of each operation within the supply chain. In the following section, we describe which information is gathered and how it is laid out within link and layout metadata.

4.1. Metaformat

To provide descriptive examples, we will adopt "canonical JSON," as described in http://wiki.laptop.org/go/Canonical_JSON, as the data format. However, applications that desire to implement in-toto are not required to use JSON. Discussion about the intended data format for in-toto can be found in the in-toto website.

4.2. File formats: general principles

All signed files (i.e., link and layout files) have the format:

```
{
  "signed" : ROLE,
  "signatures" : [
    { "keyid" : KEYID,
```

```

        "method" : METHOD,
        "sig" : SIGNATURE }
    , ... ]
}

```

Where, ROLE is a dictionary whose "_type" field describes the metadata type (as described in sections 4.3 and 4.4). KEYID is the identifier of the key signing the ROLE dictionary. METHOD is the key signing method used to generate the signature. SIGNATURE is a signature of the canonical JSON form of ROLE.

The current reference implementation of in-toto defines two signing methods, although in-toto is not restricted to any particular key signing method, key type, or cryptographic library:

- "RSASSA-PSS" : RSA Probabilistic signature scheme with appendix (<http://tools.ietf.org/html/rfc3447#page-29>). The underlying hash function is SHA256.
- "ed25519" : Elliptic curve digital signature algorithm based on Twisted Edwards curves (<http://ed25519.cr.yp.to/>).

All keys have the format:

```

{ "keytype" : KEYTYPE,
  "keyval" : KEYVAL }

```

where KEYTYPE is a string describing the type of the key, and how it is used to sign documents. The type determines the interpretation of KEYVAL.

We define two key types at present: 'rsa' and 'ed25519'.

The 'rsa' format is:

```

{ "keytype" : "rsa",
  "keyval" : { "public" : PUBLIC,
               "private" : PRIVATE }
}

```

where PUBLIC and PRIVATE are in PEM format and are strings. All RSA keys must be at least 2048 bits.

The 'ed25519' format is:

```

{ "keytype" : "ed25519",
  "keyval" : { "public" : PUBLIC,
               "private" : PRIVATE }
}

```

where PUBLIC and PRIVATE are both 32-byte strings.

Link and Layout metadata does not include the private portion of the key object:

```

{ "keytype" : "rsa",
  "keyval" : { "public" : PUBLIC }
}

```

The KEYID of a key is the hexadecimal encoding of the SHA-256 hash of the canonical JSON form of the key, where the "private" object key is excluded.

Date-time data follows the ISO 8601 standard. The expected format of the combined date and time string is "YYYY-MM-DDTHH:MM:SSZ". Time is always in UTC, and the "Z" time zone designator is attached to indicate a zero UTC offset. An example date-time string is "1985-10-21T01:21:00Z".

4.2.1 Hash object format

Hashes within in-toto are represented using a hash object. A hash object is a dictionary that specifies one or more hashes, including the cryptographic hash function. For example: { "sha256": HASH, ... }, where HASH is the sha256 hash encoded in hexadecimal notation.

4.3. File formats: layout

The layout file will be signed by a trusted key (e.g., one that was distributed beforehand) and will indicate which keys are authorized for signing the link metadata, as well as the required steps to perform in this supply chain.

The format of the layout file is as follows:

```
{ "_type" : "layout",
  "expires" : EXPIRES,
  "readme": README,
  "keys" : {
    KEYID : KEY
    , ... },
  },
  "steps" : [
    {... },
  ],
  "inspections" : [
    {... },
  ],
}
```

EXPIRES determines when layout metadata should be considered expired and no longer trusted by clients. Clients MUST NOT trust an expired file.

An optional README text can be added on the readme field. This is used to provide a human-readable description of this supply chain.

The "keys" list will contain a list of all the public keys used in the steps section, as they are described in 4.2.

The "steps" section will contain a list of restrictions for each step within the supply chain. It is also possible to delegate steps to other layouts, so that they can further specify requirements to a section of the supply chain (section 4.4.1). We will describe the contents of the steps list in section 4.3.1.

The "inspections" section will contain a list of restrictions for each step within the link. In contrast to steps, inspecting is done by the client upon verification. We will elaborate on the specifics of this process in section 4.3.2.

4.3.1 Steps

Steps performed by a functionary in the supply chain are declared as follows:

```
{
  "_name": NAME,
  "threshold": THRESHOLD,
  "expected_materials": [
    [ARTIFACT_RULE],
    ...
  ],
  "expected_products": [
    [ARTIFACT_RULE],
    ...
  ]
  "pubkeys": [
    KEYID,
    ...
  ],
  "expected_command": COMMAND,
}
```

The NAME string will be used to identify this step within the supply chain. NAME values are unique and so they MUST not repeat in different step descriptions.

The "threshold" field must contain an integer file indicating how many pieces of link metadata must be provided to verify this step. This field is intended to be used for steps that require a higher degree of trust, so multiple functionaries must perform the operation and report the same results. If only one functionary is expected to perform the step, then the "threshold" field should be set to 1.

The "expected_materials" and "expected_products" fields contain a list of ARTIFACT_RULES, as they are described in section 4.3.3. These rules are used to ensure that no materials or products were altered between steps.

The "pubkeys" field will contain a list of KEYIDs (as described in section 4.2) of the keys that can sign the link metadata that corresponds to this step.

Finally, the "expected_command" field contains a string, COMMAND, describing the suggested command to run. It is important to mention that, in a case where a functionary's key is compromised, this field can easily be forged (e.g., by changing the PATH environment variable in a host) and thus it should not be trusted for security checks. Because of this, during verification, clients should only show a warning if the expected command field does not match its counterpart in the link metadata.

It is also possible to divide a subchain by having a third-party project owner define a layout for a section of the supply chain. This can be done by means of *delegations* (as described in section 4.4.1).

4.3.2 Inspections

In contrast to steps, inspections indicate operations that need to be performed on the final product at the time of verification. For example, unpacking a tar archive to inspect its contents is an inspection step. When indicating

inspections, ARTIFACT RULES can be defined to ensure the integrity of the final product. An inspection contains the following fields.

```
{
  "_name": NAME,
  "expected_materials": [
    [ARTIFACT_RULE],
    ...
  ],
  "expected_products": [
    [ARTIFACT_RULE],
    ...
  ]
  "run": COMMAND,
  ...
}
```

Similar to steps, the NAME string will be used to identify this inspection within the supply chain. NAME values are unique and so they MUST NOT repeat in either steps or inspections.

Materials and products fields behave in the same way as they do for the steps as defined in section 4.3.1.

Finally, the "run" field contains the command to run. This field will be used to spawn a new process in the verification system to create a new piece of link metadata that correspond to the inspection steps. After running the indicated command, the materials and products fields will be verified using the artifact rules for materials and products.

4.3.2.1 Inspection interface

Executables used during an inspection need to communicate the result of running the command with in-toto. To do so, in-toto provides a simple inspection interface using the executable's return value:

- If the return value is 0, then the inspection was successful. All the artifacts and the rest of the information is recorded using the in-toto-run tool, in the same fashion as used to collect steps.
- If the result is greater than 0 and less than 127, then the inspection was not successful and validation should halt.

4.3.3 Artifact Rules

Artifact rules are used to connect steps together through their materials or products. When connecting steps together, in-toto allows the project owner to enforce the existence of certain artifacts within a step (e.g., the README can only be created in the *create-documentation* step) and authorize operations on artifacts (e.g., the compile step can use the materials from the *checkout-vcs*). The ARTIFACT_RULE format is the following:

```
{MATCH <pattern> [IN <source-path-prefix>] WITH (MATERIALS|PRODUCTS) [IN
  <destination-path-prefix>] FROM <step> ||
  CREATE <pattern> ||
  DELETE <pattern> ||
  MODIFY <pattern> ||
  ALLOW <pattern> ||
```

```
    REQUIRE <pattern> ||  
    DISALLOW <pattern>}
```

The "pattern" value is a path-pattern that will be matched against paths reported in the link metadata, including bash-style wildcards (e.g., "*"). The following rules can be specified for a step or inspection:

- **MATCH:** indicates that the files filtered in using source-path-prefix/pattern must be matched to a MATERIAL or PRODUCT from a destination step with the destination-path-prefix/pattern filter. For example, "MATCH foo WITH PRODUCTS FROM compilation" indicates that the file foo, a product of the step "compilation," must correspond to either a material or a product in this step (depending on where this artifact rule was listed). More complex uses of the MATCH rule are presented in the examples of section 5.2.

The "IN prefix" clauses are optional, and they are used to match products and materials whose path differs from the one presented in the destination step. This is the case of steps that relocate files as part of their tasks. For example MATCH foo IN lib WITH PRODUCT IN build/lib FROM compilation will ensure that the file lib/foo matches build/lib/foo from the compilation step.

- **ALLOW:** indicates that the pattern appears as material or a product of this step.
- **DISALLOW:** indicates that no files matching a pattern can appear as a material or a product on this step.
- **REQUIRE:** indicates that a pattern *must* appear as a material or product of this step.
- **CREATE:** Indicates that products matched by the pattern *must not* appear as materials of this step. Note, the rule still passes if the pattern does not match any products
- **DELETE:** indicates that materials matched by the pattern *must not* appear as products of this step. Note, the rule still passes if the pattern does not match any materials.
- **MODIFY:** indicates that the PATH's hash *must* be updated in this step. In other words, this path must appear as a material and a product within this step and their hashes must not match.

4.3.3.1 MATCH rule behavior

The MATCH rule is used to tie different steps together, by means of their materials and products. In order to ensure its correctness, it is important to describe the way it operates. To avoid any ambiguities, this will be done with the following pseudocode:

```
MATCH(source_materials_or_products_set,  
      destination_materials_or_products_set,  
      rule)  
  
# Filter source and destination materials using the rule's patterns  
source_artifacts_filtered = filter(rule.source_prefix + rule.source_pattern,  
                                   source_materials_or_products_set)  
  
destination_artifacts_filtered = filter(rule.destination_prefix + rule.destination_pattern,  
                                       destination_materials_or_products_set)
```

```

# Apply the IN clauses, to the paths, if any
for artifact in source_artifacts_filtered:
    artifact.path -= rule.source_in_clause
for artifact in destination_artifacts_filtered:
    artifact.path -= rule.destination_in_clause

# compare both sets
for artifact in source_artifacts_filtered:
    destination_artifact = find_artifact_by_path(destination_artifacts,
                                                artifact.path)

    # the artifact with this path does not exist?
    if destination_artifact == NULL:
        return FAIL

    # are the files not the same?
    if destination_artifact.hash != artifact.hash:
        return FAIL

# all of the files filtered by the source materials exist
return SUCCESS

```

Notice that if a source pattern does not match anything, verification will pass as long as no target artifact is matched either. To enforce that an artifact exists, the CREATE rule must be applied in the intended step.

4.3.3.2 Verifying expected_products and expected_materials

Rules inside the expected_products and expected_materials field are matched in the same way that a firewall behaves. Here, we describe an algorithm to illustrate the behavior of the rules being applied

```

VERIFY_EXPECTED_ARTIFACTS(rule_set, link, target_links)

# load the artifacts from the link
artifacts = load_artifacts_as_queue(link)

# iterate over all the rules
for rule in rules:
    matched_artifacts, rule_error = apply_rule(rule, artifacts)

    if rule_error:
        return ERROR("Rule failed to verify!")

    artifacts -= matched_artifacts

# check if there were any artifacts that weren't matched by the rules
if artifacts is not empty:
    return ERROR("There were unmatched artifacts in this set!")

return SUCCESS

```

This algorithm ensures that all the files have a rule that allows them to be there. Notice that these rules apply sequentially, so if there are two rules that apply to the same artifact, only the first one will be used.

4.4. File formats: [name].[KEYID-PREFIX].link

The [name].[KEYID-PREFIX].link file will contain the information recorded from the execution of a supply chain step. It lists relevant information, such as the command executed, the materials used, and the changes made to such materials (products), as well as other host information.

The name for link metadata files must contain two elements: the name of the step, and the first six bytes of the functionary's keyid separated by a dot. The KEYID portion of the name is used to avoid collisions in steps that have thresholds higher than one.

The format of the [name].[KEYID-PREFIX].link file is as follows:

```
{ "_type" : "link",
  "_name" : NAME,
  "command" : COMMAND,
  "materials": {
    PATH: HASH,
    ...
  },
  "products": {
    PATH: HASH,
    ...
  },
  "byproducts": {
    ...
  }
}
```

To identify to which step a piece of link metadata belongs, the NAME field must be set to the same identifier as the step described in the layout, as specified in section 4.3.1.

The COMMAND field contains the command and its arguments as executed by the functionary.

The "materials" and "products" fields are dictionaries keyed by a file's PATH. Each HASH value is a hash object as described in section 4.2.1.

The "byproducts" field is an opaque dictionary that contains additional information about the step performed. Byproducts are not verified by in-toto's default verification routine. However, the information gathered can be used for further scrutiny during an inspection step.

4.4.1 Delegating to other layouts

It is possible that a project owner cannot define all the steps at the appropriate level of detail. Such a case might occur with a software project that statically compiles a third-party library. When this is the case, additional layouts can be used to describe parts of the supply chain with more granularity. We call these additional layouts *delegations*.

To create a delegation, a series of steps are declared, and thus the functionary will take the role of a third party project owner. Delegated [name].[hashprefix].link files will have the format described for a layout file (section 4.3) instead of the usual contents of a link metadata file.

The signature of this new layout file must match the one the top-level project owner intended for this step.

Inspections in delegated layouts can be verified in two ways:

- A piece of signed link metadata is presented for the inspection. The key to sign for this piece of link metadata must be the same key used to sign the delegated layout file.
- If the piece of link metadata is not present, the client must perform that inspection upon validation. When this is the case, the inspection must be run while validating the delegated layout. This means when the verification algorithm recurses inside the delegated layout.

5. Detailed workflows

To provide further detail of in-toto's workflow, we describe a detailed case that includes most of the concepts explained in sections 3 and 4. This is an error-free case.

1. The project owner defines the layout to be followed by, e.g. using the in-toto CLI tools. When doing so, he or she instructs who is intended to sign for every piece of link metadata, any delegations that may exist, and how to further verify accompanying metadata.
2. Functionaries perform the intended actions and produce link metadata for each step.
3. Once all the steps are performed, the final product is shipped to the client.
4. Toto's verification tools are run on the final product.
5. Toto inspects the final product to find a root.layout file that describes the top-level layout for the project. The signature(s) on the file are checked using previously-acquired project owner public key(s). Subsequently, if the layout signature verification passes, the functionaries' public keys are loaded from the layout.
6. The expiration time is verified to ensure that this layout is still fresh.
7. The steps, as defined in the layout, are loaded. For each step in the layout, one or more pieces of link or layout metadata is loaded.
 - a. If the file loaded metadata is a link metadata file, a data structure containing the materials and products is populated with the reported values.

If more than one piece of link metadata are loaded due to the threshold value, their materials and products fields must exactly match.
 - b. If the file is a layout file instead, the algorithm will recurse into that layout, starting from step 5.

- c. If there is a layout file in conjunction with a link file (i.e., one of the functionaries made a delegation while others didn't), then verification should fail.
8. Inspection steps are executed, and the corresponding materials and products data structures are populated.
9. Matching rules are applied against the products and materials reported by each step as described by the algorithm in section 4.3.3.2.
10. If all these steps are successful, verification is done. The user can now use the final product.

5.2 Supply chain examples

To better understand how in-toto works, we provide a series of examples of the framework being used in practice. We will start with a simple example to showcase how the relevant aspects of in-toto come into play. After this, we will present a more complete and realistic example. Additional link and layout metadata examples can be found at in-toto.io.

5.2.1 Alice's Python script

This first example covers a simple scenario to show in-toto's elements without focusing on the details of a real-life supply chain.

In this case, Alice writes a Python script (foo.py) using her editor of choice. Once this is done, she will provide the script to her friend, Bob, who will package the script into a tarball (foo.tar.gz). This tarball will be sent to the client, Carl, as part of the final product.

When providing Carl with the tarball, Alice's layout tells Carl's installation script that it must make sure of the following:

- That the script was written by Alice herself
- That the packaging was done by Bob
- Finally, since Bob is sometimes sloppy when packaging, Carl must also make sure that the script contained in the tarball matches the one that Alice reported on the link metadata.

As a result of this, Alice's layout would have two steps and one inspection.

A root.layout file that fulfills these requirements would look like this:

```
{ "signed" : {
  "_type" : "layout",
  "expires" : EXPIRES,
  "keys" : {
    BOBS_KEYID : BOBS_PUBKEY
    ALICES_KEYID: ALICES_PUBKEY
  },
  "steps" : [
    {
```

```

    "_name": "write-code",
    "threshold": 1,
    "expected_materials": [ ],
    "expected_products": [
        [ "CREATE", "foo.py" ] ],
    ],
    "pubkeys": [
        ALICES_KEYID,
    ],
    "expected_command": "vi",
},
{
    "_name": "package",
    "threshold": 1,
    "expected_materials": [
        [ "MATCH", "foo.py", "WITH", "PRODUCTS", "FROM", "write-code" ] ],
    ],
    "expected_products": [
        [ "CREATE", "foo.tar.gz" ] ],
    ],
    "pubkeys": [
        "BOBS_KEYID",
    ],
    "expected_command": "tar zcvf foo.tar.gz foo.py",
}]
"inspect": [
    {
        "_name": "inspect_tarball",
        "expected_materials": [
            [ "MATCH", "foo.tar.gz", "WITH", "PRODUCTS", "FROM", "package" ] ],
        ],
        "expected_products": [
            [ "MATCH", "foo.py", "WITH", "PRODUCTS", "FROM", "write-code" ] ],
        ],
        "run": "inspect_tarball.sh foo.tar.gz",
    }
],
},
"signatures" : [
    { "keyid" : ALICES_KEYID,
      "method" : "ed25519",
      "sig" : "90d2a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eae
f4df84891d5aa37ace3ae3736a698e082e12c300dfe5aee92ea33a8f461f02"
    }
]
}

```

From this layout file, we can see that Alice is expected to create a foo.py script using vi. The signed link metadata should be done with Alice's key (for simplicity, the same key is used to sign the layout and the first link metadata). After this, Bob is expected to use "tar zcvf ..." to create a tarball, and ship it to Carl. We assume that Carl's machine already hosts an inspect_tarball.sh script, which will be used to inspect the contents of the tarball.

After both steps are performed, we expect to see the following pieces of link metadata:

write-code.[ALICE-KEYID-PREFIX].link:

```

{"signed" : {
  "_type" : "link",
  "name": write-code,
  "command" : "vi foo.py",
  "materials": { },
  "products": {
    "foo.py": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d", }
  },
  "return-value": "0",
  "byproducts": { }
}
"signatures" : [
  { "keyid" : ALICES_KEYID,
    "method" : "ed25519",
    "sig" :
      "94df84890d7ace3ae3736a698e082e12c300dfe5aee92ea33a8f461f
      022a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eae1d5aa3"
    }
]
}

```

package.[BOB-KEYID-PREFIX].link:

```

{"signed" : { "_type" : "link",
  "Name": "package",
  "command" : "tar zcvf foo.tar.gz foo.py",
  "materials": {
    "foo.py": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d", }
  },
  "products": {
    "foo.tar.gz": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b", }
  },
  "return-value": "0",
  "byproducts": {
    "stdout": "foo.py",
  }
}
"signatures" : [
  { "keyid" : BOBS_KEYID,
    "method" : "ed25519",
    "sig" :
      "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a06c7a6c2
      a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eae1d5aa394df84890d7ace3"
    }
]
}

```

With these three pieces of metadata, along with foo.tar.gz, Carl can now perform verification and install Alice's foo.py script.

When Carl is verifying, his installer will perform the following checks:

1. The root.layout file exists and is signed with a trusted key (in this case, Alice's).
2. Every step in the layout has a corresponding [name].link metadata file signed by the intended functionary.

3. All the matching rules on every step match the rest of the [name].link metadata files.

Finally, inspection steps are run on the client side. In this case, the tarball will be extracted using `inspect_tarball.sh` and the contents will be checked against the script that Alice reported in the link metadata.

If all of these verifications pass, then installation continues as usual.

5.2.2 Alice uses a version control system

This time, Alice uses a Version Control System (VCS) to ease collaboration with Diana, a new developer on the team. Since efficiency is now an issue, Alice also ported her Python script to C. Lastly, a third-party (Eleanor) has been assigned to compile the binary.

In this case, Alice and Diana commit to their central repository, and update their source file (`src/foo.c`). When all the milestones are met, Alice --- and only Alice --- will tag the sources and send them to Eleanor for compilation.

When Eleanor receives the tarball, she is required to compile the source code into the binary (`foo`), and send it over to Bob, who is still in charge of packaging. Bob will package and send `foo.tar.gz` to Carl, so he can install and use the product.

When providing Carl with the tarball, Alice's layout tells Carl that he must make sure of the following:

- That the source code was written by Alice and Diana
- That tagging was done by Alice
- That compilation was done by Eleanor
- That Eleanor's binary was packaged by Bob

Finally, since Bob is sometimes sloppy when packaging, Carl must also make sure that the binary contained in the tarball matches the one that Eleanor reported at the end of her step. Also, in order to ensure that the VCS was only altered by Diana and Alice, an additional piece of metadata (a signed log) is provided that Carl will inspect to ensure it is correct. Both of these operations are to be done when performing inspection steps.

As a result of this, Alice's layout would have three steps and two inspections.

A `root.layout` file that fulfills these requirements would look like this:

```
{ "signed" : { "_type" : "layout",
  "expires" : EXPIRES,
  "keys" : {
    BOBS_KEYID : BOBS_PUBKEY,
    DIANAS_KEYID : DIANAS_PUBKEY,
    ELEANORS_KEYID : ELEANORS_PUBKEY,
    ALICES_KEYID : ALICES_PUBKEY,
  },
  "steps" : [
```

```

{
  "_name": "checkout-vcs",
  "threshold": 1,
  "expected_materials": [ ],
  "expected_products": [
    ["CREATE", "src/foo.c"],
    ["CREATE", "vcs.log"],
  ],
  "pubkeys": [
    ALICES_KEYID,
  ],
  "expected_command": "git tag",
},
{
  "_name": "compilation",
  "threshold": 1,
  "expected_materials": [
    ["MATCH", "src/foo.c", "WITH", "PRODUCTS", "FROM", "checkout-vcs"]
  ],
  "expected_products": [
    ["CREATE", "foo"],
  ],
  "pubkeys": [
    "ELEANORS_KEYID",
  ],
  "expected_command": "gcc -o foo src/foo.c",
},
{
  "_name": "package",
  "threshold": 1,
  "expected_materials": [
    ["MATCH", "foo", "WITH", "PRODUCTS", "FROM", "compilation"],
  ],
  "expected_products": [
    ["CREATE", "foo.tar.gz"],
  ],
  "pubkeys": [
    "BOBS_KEYID",
  ],
  "expected_command": "tar -zcvf foo.tar.gz foo",
}
],
"inspect": [
  {
    "_name": "check-package",
    "expected_materials": [
      ["MATCH", "foo.tar.gz", "WITH", "PRODUCTS", "FROM", "package"]
    ],
    "expected_products": [
      ["MATCH", "src/foo.c", "WITH", "PRODUCTS", "FROM", "checkout-vcs"],
    ],
    "run": "inspect_tarball.sh foo.tar.gz",
  },
  {
    "_name": "verify-vcs-commits",
    "expected_materials": [
      ["MATCH", "vcs.log", "WITH", "PRODUCTS", "FROM", "checkout-vcs"]
    ],
  },

```

```

    ],
    "expected_products": [
      ["MATCH", "src/foo.c", "WITH", "PRODUCTS", "FROM", "checkout-vcs"],
    ],
    "run": "inspect_vcs_log -l vcs.log -P ALICES_PUBKEY -P DIANAS_PUBKEY",
  }
],

},
"signatures" : [
  { "keyid" : ALICES_KEYID,
    "method" : "ed25519",
    "sig" : "90d2a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaeed
f4df84891d5aa37ace3ae3736a698e082e12c300dfe5aee92ea33a8f461f02"
  }
]
}

```

In contrast to the previous example, Carl will perform two different inspections on the final product. While the inspection of the tarball will be performed in the same manner as in the previous example, there is a new inspection phase that must verify a signed log from the VCS.

Because of this, the `vcs.log` is listed as a product from the first step. It will be used to verify the operations within the VCS by executing a VCS-specific script. It is worth noting, however, that the relevant public keys are passed as arguments to the script, for the script and its parameters are agnostic to in-toto.

When the three steps are carried out, we expect to see the following pieces of link metadata:

checkout-vcs.link:

```

{"signed" : { "_type" : "link",
  "name": "checkout-vcs",
  "command" : "git tag 1.0",
  "materials": { },
  "products": {
    "src/foo.c": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d", },
    "vcs.log": { "sha256": "e64589ab156f325a4ab2bc5d532737d5a7"
  },
  "return-value": "0",
  "byproducts": { }
}
"signatures" : [
  { "keyid" : ALICES_KEYID,
    "method" : "ed25519",
    "sig" :
      "94df84890d7ace3ae3736a698e082e12c300dfe5aee92ea33a8f461f
      022a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaeedf1d5aa3"
    }
]
}

```

compilation.link:

```

{"signed" : { "_type" : "link",
  "name": "compilation",

```

```

"command" : "gcc -o foo foo.c",
"materials": {
  "src/foo.c": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d",}
},
"products": {
  "foo": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b",}
},
"return-value": "0",
"byproducts": {
  "stdout": "",
}
}
"signatures" : [
  { "keyid" : ELEANORS_KEYID,
    "method" : "ed25519",
    "sig" :
      "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a06c7a6c2
      a6a93a9f5771eb2e5ce0c93dd580bec2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
    }
]
}

```

package.link:

```

{"signed" : { "_type" : "link",
  "name": "package",
  "command" : "tar zcvf foo.tar.gz foo",
  "materials": {
    "foo": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b",}
  },
  "products": {
    "foo.tar.gz": { "sha256": "f73c9cd37d8a6e2035d0eed767f9cd5e",}
  },
  "return-value": "0",
  "byproducts": {
    "stdout": "foo",
  }
}
"signatures" : [
  { "keyid" : BOBS_KEYID,
    "method" : "ed25519",
    "sig" :
      "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a06c7a6c2
      a6a93a9f5771eb2e5ce0c93dd580bec2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
    }
]
}

```

With these three pieces of metadata, along with foo.tar.gz, Carl can now perform verification as we described in the previous example.

5.2.2 Alice delegates to a third party

A common scenario in software distributions is that source code is produced upstream. For example, in Linux distributions, the source code for bash (the default shell in many distributions) is written by the Free Software

Foundation. Given this, the task of writing the source code, is delegated to the third-party, often referred to as "upstream."

In this example, the maintainer will delegate the "write-code" task to upstream, while the compilation and package steps will be performed by local contributors to the project. For brevity, we will skip the pieces of link metadata that are relevant to the delegation. In addition, inspections on the top-level layout will be omitted to simplify both layouts.

A root.layout file that fulfills these requirements would look like this:

```
{ "signed" : { "_type" : "layout",
  "expires" : EXPIRES,
  "keys" : {
    BOBS_KEYID : BOBS_PUBKEY
    DIANAS_KEYID : DIANAS_PUBKEY
    ELEANORS_KEYID : ELEANORS_PUBKEY
    UPSTREAM_KEYID: UPSTREAM_KEYID
  },
  "steps" : [
    {
      "_name": "fetch-upstream",
      "threshold": 1,
      "expected_materials": [ ],
      "expected_products": [
        ["CREATE", "src/*"],
      ],
      "pubkeys": [
        UPSTREAM_KEYID,
      ],
      "expected_command": "",
    },
    {
      "_name": "compilation",
      "threshold": 1,
      "expected_materials": [
        ["MATCH", "src/*", "WITH", "PRODUCTS", "FROM", "fetch-upstream"],
      ],
      "expected_products": [
        ["CREATE", "foo"],
      ],
      "pubkeys": [
        "ELEANORS_KEYID",
      ],
      "expected_command": "gcc -o foo src/*",
    },
    {
      "_name": "package",
      "threshold": 1,
      "expected_materials": [
        ["MATCH", "foo", "WITH", "PRODUCTS", "FROM", "compilation"],
      ],
      "expected_products": [
        ["CREATE", "foo.tar.gz"],
      ],
      "pubkeys": [
        "BOBS_KEYID",
      ],
    },
  ],
}
```

```

        ],
        "expected_command": "tar -zcvf foo.tar.gz foo",
    }
],
"inspect": [],

},
"signatures" : [
    { "keyid" : ALICES_KEYID,
      "method" : "ed25519",
      "sig" : "90d2a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaeed
f4df84891d5aa37ace3ae3736a698e082e12c300dfe5aee92ea33a8f461f02"
    }
]
}

```

fetch-upstream.link:

```

{"signed" : { "_type" : "layout",
  "expires" : EXPIRES,
  "keys" : {
    UPSTREAM_DEV1_KEYID: UPSTREAM_DEV1_KEY,
    UPSTREAM_DEV2_KEYID: UPSTREAM_DEV2_KEY,

  },
  "steps" : [
    {
      "_name": "checkout-vcs",
      "threshold": 1,
      "expected_materials": [ ],
      "expected_products": [
        ["CREATE", "src/*"],
        ["CREATE", "vcs.log"]
      ],
      "pubkeys": [
        UPSTREAM_DEV1_KEYID,
      ],
      "expected_command": "git tag -s",
    },
    {
      "_name": "compile-docs",
      "threshold": 1,
      "expected_materials": [
        ["MATCH", "src/*", "WITH", "PRODUCTS", "FROM", "check-out-vcs"],
      ],
      "expected_products": [
        ["CREATE", "doc/*"],
      ],
      "pubkeys": [
        "UPSTREAM_DEV2_KEYID",
      ],
      "expected_command": "sphinx",
    },
    {
  ],

```

```

    "inspect": [
      {
        "_name": "verify-vcs-commits",
        "expected_materials": [
          ["MATCH", "vcs.log", "WITH", "PRODUCTS", "FROM", "check-out-vcs"]
        ],
        "expected_products": [
          ["MATCH", "src/*", "WITH", "PRODUCTS", "FROM", "check-out-vcs"],
        ]
        "run": "inspect_vcs_log -l vcs.log -P UPSTREAM_PUBKEY -P UPSTREAM_PUBKEY",
      }
    ],

  },
  "signatures" : [
    { "keyid" : UPSTREAM_KEYID,
      "method" : "ed25519",
      "sig" : "90d2a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eae
d4df84891d5aa37ace3ae3736a698e082e12c300dfe5aee92ea33a8f461f02"
    }
  ]
}

```

check-out-vcs.link:

```

{"signed" : { "_type" : "link",
  "name": "compilation",
  "command" : "gcc -o foo foo.c",
  "materials": {
  },
  "products": {
    "src/foo.c": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d",}
    "vcs.log": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b",}
  },
  "return-value": "0",
  "byproducts": {
    "stdout": "",
  }
}
"signatures" : [
  { "keyid" : UPSTREAM_DEV1_KEYID,
    "method" : "ed25519",
    "sig" :
      "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a06c7a6c2
a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eae
df1d5aa394df84890d7ace3"
  }
]
}

```

compile-docs.link:

```

{"signed" : { "_type" : "link",
  "name": "package",
  "command" : "tar zcvf foo.tar.gz foo",
  "materials": {
    "foo": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b",}
  },
  "products": {

```

```

        "foo/doc/index.html": { "sha256": "f73c9cd37d8a6e2035d0eed767f9cd5e"},
    },
    "return-value": "0",
    "byproducts": { }
}
"signatures" : [
    { "keyid" : UPSTREAM_DEV2_KEYID,
      "method" : "ed25519",
      "sig" :
        "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a06c7a6c2
        a6a93a9f5771eb2e5ce0c93dd580bec2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
    }
]
}

```

verify-vcs-commits.link (upstream inspection):

```

{"signed" : { "_type" : "link",
  "name": "package",
  "command" : "inspect_vcs_log -l vcs.log -P UPSTREAM_PUBKEY -P
    UPSTREAM_PUBKEY",
  "materials": {
    "vcs.log": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b"},
  },
  "products": {
    "foo/foo.c": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b"},
  },
  "return-value": "0",
  "byproducts": { }
}
"signatures" : [
    { "keyid" : UPSTREAM_DEV2_KEYID,
      "method" : "ed25519",
      "sig" :
        "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a06c7a6c2
        a6a93a9f5771eb2e5ce0c93dd580bec2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
    }
]
}

```

compilation.link:

```

{"signed" : { "_type" : "link",
  "name": "compilation",
  "command" : "gcc -o foo foo.c",
  "materials": {
    "src/foo.c": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d"},
  },
  "products": {
    "foo": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b"},
  },
  "return-value": "0",
  "byproducts": {
    "stdout": "",
  }
}
"signatures" : [
    { "keyid" : ELEANORS_KEYID,

```

```

        "method" : "ed25519",
        "sig" :
        "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a06c7a6c2
        a6a93a9f5771eb2e5ce0c93dd580bec2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
    }
}

```

package.link:

```

{"signed" : { "_type" : "link",
  "name": "package",
  "command" : "tar zcvf foo.tar.gz foo",
  "materials": {
    "foo": { "sha256": "78a73f2e55ef15930b137e43b9e90a0b", }
  },
  "products": {
    "foo.tar.gz": { "sha256": "f73c9cd37d8a6e2035d0eed767f9cd5e", }
  },
  "return-value": "0",
  "byproducts": {
    "stdout": "foo",
  }
}
"signatures" : [
  { "keyid" : BOBS_KEYID,
    "method" : "ed25519",
    "sig" :
    "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a06c7a6c2
    a6a93a9f5771eb2e5ce0c93dd580bec2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
  }
]
}

```

From this example, we can see that the "fetch-upstream" link is actually a layout file that further defines two steps: check-out-vcs and compile docs. These two steps are performed and verified in the same way as the top-level link metadata.

Notice also, that the inspection is run by the upstream developer, and produces link metadata that is signed with the same key as the layout. If this piece of link metadata was missing, the client would be in charge of running the inspection (which would require the vcs.log file to be shipped along with the final product).

6 Learn More

This document introduced in-toto, a framework to secure the integrity of software supply chains. Although minimal, this framework aims to be easily extensible to include further restrictions (by means of inspections) that are specific to each supply chain. This allows the framework to offer a large degree of flexibility to meet varied applications.

We invite all interested parties to test our reference implementation of in-toto [here](#), take a look at our examples, or read more about in-toto [here](#).