Toto Specification

Sept 28, 2016

Version 0.1c

1. Introduction

1.1. Scope

This document describes Toto, a system for securing the way in which software is developed, built, tested, and packaged (i.e., the software supply chain). This includes providing integrity and verifiability to all the actions performed while writing code, compiling, testing, and deploying software.

Toto is designed to ensure the integrity of a software product from initiation to end-user installation. 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, Toto allows the user to verify if a step in the supply chain was intended to be performed, and if the step was performed by the right actor.

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 it is susceptible to these threats, supply chain breaches are 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. A typical software supply chain is composed of multiple steps "chained" together that transform (e.g., compilation) or verify the state (e.g., linting) of the project in order to drive it to a final product.

1.5. Goals

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 a software product.

1.5.1 Goals for implementation

- The end-user side of the framework must be straightforward to implement in any programming language and for any platform with the requisite cryptographic support.
- The framework should be easy to integrate with existing installers, build systems, testing frameworks, VCSes, 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.

1.5.1 Defender Goals and Non-goals

In the context of Toto, a defender is the end user (i.e., the person who will install the software product), as well as actors performing steps in the supply chain (functionaries) and the actors in charge of a software product (or project owner). With this in mind, we describe the following goals for defenders.

Goals:

The user's goal is to install a target software product that has gone through all the required 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 all code must be reviewed by two developers before being committed, Toto will verify that the git commit log indicates two developers reviewed the code. We assume that there will not be two colluding (or deceived) developers who jointly introduce a vulnerability.

We also assume the defender has configured the layout 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 Toto's layout so that no code review is performed and a package is built on an untrusted server --- which is an incredibly unsafe layout. While this is visible to users installing the software, Toto's role is not to judge which layouts are

insecure and then to block such layouts. However, tools that integrate into Toto may independently block or make judgments about the security of a specific layout.

1.5.2. Assumptions

The end-user side tools should perform key-management on behalf of the user. The end-user is never required to retrieve and provide keys for verification.

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

The root of trust will not rely on external Public Key Infrastructure (PKI). That is, no authority will be derived from keys outside of the framework.

1.5.3 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 end user 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 end user followed the layout specified by the project owner. All steps described were executed in the right order, and, if audited by a third party, he or she can verify that all steps were performed as described. For example, within the 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 were performed can be identified, as well as the materials used and resulting products.

- **Step authentication:** the actor who carried 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 end-user systems that further verify the integrity of each step.

1.6. Terminology

- **Software supply chain** (or SCC): the series of actions performed to create a software product. These steps usually begin with users committing to the version control system and end with the software product's installation on the end-user's system.

- **Supply chain layout** (or simply layout): dictates the series of steps that need to be carried out in the SCC to create the 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 (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).

- **Products:** 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 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 end user's system. This includes 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 end user'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.

- **Target files:** the files that will be installed in the end-user's system. In the case of an installer, these will often be unpacked from within the final product and made ready for use on the user’s system.

2. System overview

Toto's main goal is to provide authentication and integrity guarantees of the supply chain that created the final product that the client will install.

To avoid ambiguity, we will refer to any files in the final product that 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 Toto.

A 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 end user's machine.

2. Each functionary performs their 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 end user 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, and that each step was performed by the right 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 end-user 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 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 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:** installs and uses the final product. Before doing so, the client will perform verification, by checking the the layout file is signed by a trusted owner and then contrasting the steps described in the layout with the corresponding links, and carrying out the inspection steps.

In addition, there are third-party equivalents of the above roles, which are managed by the delegation mechanism, described in section 2.1.3.

2.1.1 Project Owner

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.

The project owner can delegate parts of the supply chain to third-parties. In this case, a subset of the steps to be performed are defined by the delegated project owner.

We assume that the client-side of the framework already has knowledge of the key used by each project owner to sign layout metadata.

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.

2.1.3 Clients

Clients are end users who want to use the product.

The end user will perform verification on the final product. This includes verifying the layout metadata, and that the link metadata provided matches what the specified layout metadata describes, and performing inspection steps to ensure that any additional metadata and target files meet the specified criteria.

A client will likely introduce the Toto framework 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, he/she will allow a third party to define the series of steps required. This is helpful if the project owner does not know the specifics of a step, but trusts a functionary to specify them on the run 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 Toto Components

A 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.
- A tool that functionaries can use to create link metadata about a step.
- A tool to be used by the end user 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 verification steps, as directed by the layout. This tool may often be included in a package manager or installer.

While all of these tools could be independent implementations, it is expected that sharing cryptographic and parsing libraries may benefit a system implementer.

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 end user, Carl, as part of the final product. Carl's target file (i.e., the file he wants to run) is foo.py

When providing Carl with the tarball, 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 a corresponding link metadata file signed by the intended functionary, as described in the layout.
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.
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.

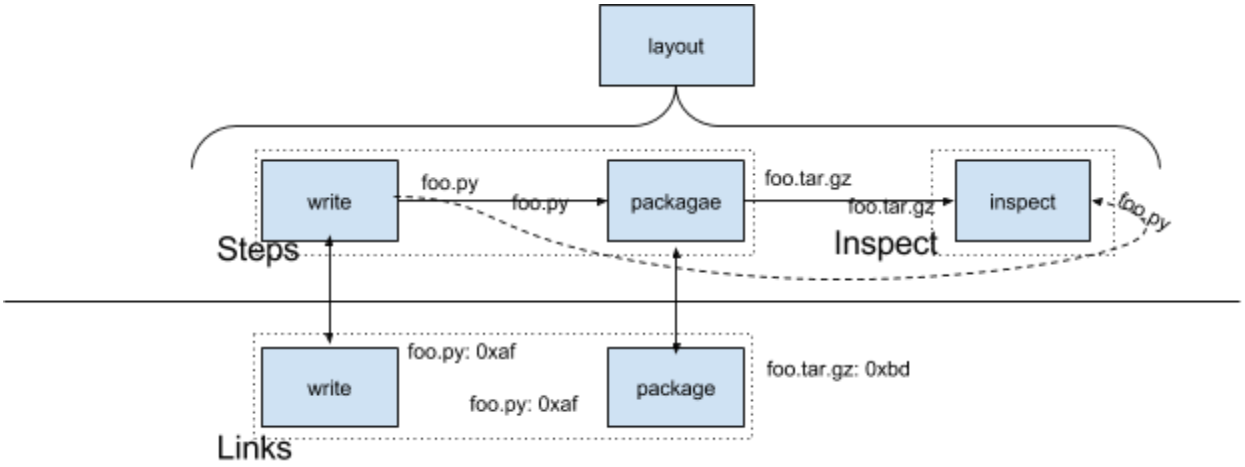

Figure 1: the pieces of the supply chain for this example

3. The final product

The final product is the bundle of link and layout metadata and target files that will be used by the end-user's system.

An installer or package manager uses the framework to inspect the final product to 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 Toto to validate 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 metadata for verification.

3.1.1 Supply Chain Layout

The supply chain layout specifies each of the different steps and the requirements for each step, as well as the public keys used by functionaries to perform 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, when they are altered, it is only by an intended functionary.

3.1.3 Target files

Target files are the files end-users 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 it.

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 the specifics of each operation within the supply chain. We describe which information is gathered and how it is laid out within link and layout metadata next.

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 Toto are not required to use JSON. Discussion about the intended data format for Toto can be found on our website.

4.2. File formats: general principles

All signed files have the format:

```
{ "signed" : ROLE,
  "signatures" : [
     { "keyid" : KEYID,
       "method" : METHOD,
       "sig" : SIGNATURE }
     , ... ]
}
```

Where, ROLE is a dictionary whose "_type" field describes the role type (as described in section 2.1). 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 Toto defines two signing methods, although Toto is not restricted to any particular key signing method, key type, or cryptographic library:

"RSASSA-PSS" : RSA Probabilistic signature scheme with appendix.
The underlying hash function is SHA256.

"ed25519" : Elliptic curve digital signature algorithm based on
Twisted Edwards curves.

RSASSA-PSS: http://tools.ietf.org/html/rfc3447#page-29 ed25519: 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 keytypes 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 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,
   "keys" : {
       KEYID : KEY
       , ... },
   },
   "steps" : [
       {... },
   ],
   "inspect" : [
       {... },
   ],
}
```

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

The "keys" list will contain a list of all the public keys used in the steps section.

The "steps" section will contain a list of restrictions for each step within the supply chain. Steps are the operations in the supply chain performed by functionaries. 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 "inspect" section will contain a list of restrictions for each step within the link. In contrast to steps, inspecting is done by the end user upon installation. We will elaborate on the specifics of this in section 4.3.2.

4.3.1 Steps

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

```
{
  "_name": NAME,
  "expected_materials": [
     [MATCHRULE],
     ...
  ],
  "expected_products": [
     [MATCHRULE],
     ...
  ]
  "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 "materials" and “products” fields contain a list of MATCHING 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 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.

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 on section 4.4.1).

4.3.2 Inspections

In contrast to links, 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, restrictions can be defined to ensure the integrity of the final product. An inspection contains the following fields.

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

Similar to steps, 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 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 a suggested command to run. This field will be used to spawn a new process in the verification system to verify a piece of link metadata separately. After running the indicated command, the materials and products fields will be verified using the matching rules for materials and products.

4.3.3 Matching Rules

Matching rules indicate the sources for certain materials and products. The MATCHRULE format is the following:

```
{MATCH (PRODUCT|MATERIAL) <path> FROM <step> ||
            CREATE <path> ||
            DELETE <path> ||
             MODIFY <path>}
```

The PATH value is a filepath that will be reported by the verifier, including bash-style wildcards (e.g., “*”). The OPERATION field indicates which action is to be verified in the indicated PATH. The OPERATION field must be one of the following keywords:

- **MATCH:** indicates that the PATH must be matched to a material or product from another step. For example, "MATCH product foo 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 matching rule was listed).

- **CREATE/DELETE**: indicates that the PATH must be recorded or tracked for this step. If the link metadata does not contain the hash of this file as a material or product (as indicated), verification will fail.

- **MODIFY:** indicates that the PATH's hash will be updated in this step.

4.4. File formats: [name].link

The [name].link file will contain the information recorded from the execution of a link. 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 format of the name.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 the functionary executed the command, or "EDIT" a string indicating an edit event. When “EDIT” is being used, the command that was run is considered unknown; Editing is useful when the command executed cannot be gathered or wrapped by Toto-compliant libraries.

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 Toto’s default verification routine. However, the information gathered can be used for further inspection 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].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.

5. Detailed Workflows

To provide further detail of Toto’s workflow, we describe a detailed workflow that includes all 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 Toto cli tools. When doing so, he 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 end user.

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 on the file is checked using a previously-acquired public key. Additional public keys are loaded from the layout if the signature verification passes.

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 element in the layout, a piece of link or layout metadata is loaded.
    a. If the file loaded is a link metadata file, a data structure containing the materials and products is populated with the reported values.
    b. If the file is a layout file instead, the algorithm will recurse into that layout, starting from step 5.

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. If a required matching rule fails to verify, the whole process is halted and the user is notified.

10. If all these steps are successful, installation is carried out as usual.

5.2 Supply chain examples

To better understand how 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 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 URL

5.2.1 Alice's Python script

This first example covers a simple scenario to show 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 end user, 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",
         "expected_materials": [ ],
         "expected_products": [
            [["CREATE", "foo.py"]],
         ]
         "pubkeys": [
            ALICES_KEY,
         ],
         "expected_command": "vi",
       },
       {
         "_name": "package",
         "expected_materials": [
            [["MATCH", "PRODUCT", "foo.py", "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", "PRODUCT", "foo.tar.gz", "FROM", "package"]]
         ],
         "expected_products": [
            [["MATCH", "PRODUCT", "foo.py", "FROM", "write-code"]],
         ]
         "run": "inspect_tarball.sh foo.tar.gz",
       }
    ],

    },
    "signatures" : [
        { "keyid" : ALICES_KEYID,
          "method" : "ed25519",
          "sig" : "90d2a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaed
         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.link:

```
{"signed" : { "_type" :  "link",
       "command" : "vi foo.py",
       "materials": { },
       "products": {
          "foo.py": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d",}
       },
       "return-value": "0",
       "byproducts": { }
     }
"signatures" : [
        { "keyid" : ALICES_KEYID,
          "method" : "ed25519",
          "sig" :
          "94df84890d7ace3ae3736a698e082e12c300dfe5aee92ea33a8f461f
          022a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaedf1d5aa3"
          }
        ]
}
```

package.link:

```
{"signed" : { "_type" :  "link",
       "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
          a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
          }
        ]
}
```

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 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 script 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 script contained in the tarball matches the one that Alice 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. These both 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
           ELEANOR_KEYID : ELEANOR_PUBKEY
           ALICES_KEYID: ALICES_PUBKEY
        },
       "steps" : [
            {
              "_name": "checkout-vcs",
              "expected_materials": [ ],
```

```
      "expected_products": [
         [["CREATE", "src/foo.c"],
          ["CREATE", "vcs.log"],
      ]
      "pubkeys": [
         ALICES_KEY,
      ],
      "expected_command": "git tag *",
    },
    {
      "_name": "compilation",
      "expected_materials": [
         [["MATCH", "PRODUCT", "src/foo.c", "FROM",
         "checkout-vcs"]],
      ],
      "expected_products": [
         [["CREATE", "foo"]],
      ]
      "pubkeys": [
         "ELEANORS_KEYID",
      ],
      "expected_command": "gcc -o foo src/foo.c",
    },
    {
      "_type": "step",
      "_name": "packaging",
      "expected_materials": [
         [["MATCH", "PRODUCT", "foo", "FROM",
         "compilation"]],
      ],
      "expected_products": [
         [["CREATE", "foo.tar.gz"]],
      ]
      "pubkeys": [
         "BOBS_KEYID",
      ],
      "expected_command": "tar -zcvf foo.tar.gz foo",
    }
],
"inspect": [
    {
      "_name": "check-package",
      "materials": [
         [["MATCH", "PRODUCT", "foo.tar.gz", "FROM", "package"]]
      ],
      "products": [
         [["MATCH", "PRODUCT", "src/foo.c", "FROM", "write-code"],
      ]
      "run": "inspect_tarball.sh foo.tar.gz",
    }
    {
      "_name": "verify-vcs-commits",
      "materials": [
         [["MATCH", "PRODUCT", "vcs.log", "FROM", "package"]]
      ],
      "products": [
         [["MATCH", "PRODUCT", "src/foo.c", "FROM",
```

```
                    "checkout-vcs"],
                  ]
                  "run": "inspect_vcs_log -l vcs.log -P ALICES_PUBKEY -P
                  DIANAS_PUBKEY",
                }
          ],

      },
"signatures" : [
       { "keyid" : ALICES_KEYID,
         "method" : "ed25519",
         "sig" : "90d2a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaed
          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 Toto.

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

checkout-vcs.link:

```
{"signed" : { "_type" :  "link",
       "command" : "git tag 1.0",
       "materials": { },
       "products": {
          "src/foo.c": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d",}
       },
       "return-value": "0",
       "byproducts": { }
     }
"signatures" : [
       { "keyid" : ALICES_KEYID,
         "method" : "ed25519",
         "sig" :
         "94df84890d7ace3ae3736a698e082e12c300dfe5aee92ea33a8f461f
         022a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaedf1d5aa3"
         }
       ]
}
```

compilation.link:

```
{"signed" : { "_type" :  "link",
       "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
          a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
          }
        ]
}
```

package.link:

```
{"signed" : { "_type" :  "link",
        "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
          a6a93a9f5771eb2e5ce0c93dd580bebc2080d10894623cfd6eaedf1d5aa394df84890d7ace3"
          }
        ]
}
```

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

6 Conclusion

This document introduced 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. As a benefit of this, the framework offers a large degree of flexibility.

We invite all the interested parties to test our reference implementation of Toto here, take a look at our examples, or read more about Toto here.