Software Supply Chain Improvements to TUF Using in-toto

Athena C. Hernandez

Secure Systems Lab, Tandon School of Engineering

New York University GSTEM Program

Justin Cappos, Lukas Pühringer, Aditya Sirish

August 15, 2022

Abstract

The overall objective of the following study is to simplify the implementation of the in-toto software for Python-TUF developers. I approached this by creating a Python script file for part of the verification instructions of in-toto. This way, developers only need to run one line into their command prompt versus the current surfeit of instructions that are provided in a Markdown file on TUF's GitHub repository. Both in-toto and TUF are used to secure the vulnerabilities between steps in the supply chain. However, most developers overlook this part because of how difficult implementation can be for these softwares. Automating part of the verification process gives software developers more incentive to incorporate software supply chain tools into their work. Recently, software supply chains have been exposed to attacks in various ways by hackers—especially the updating process that TUF takes care of. Some attacks include arbitrary software installation, rollback attacks, fast-forward attacks, indefinite freeze attacks, endless data attacks, extraneous dependency attacks, mix-and-match attacks, wrong software installation, malicious mirrors, and vulnerabilities to key compromises.

Keywords: software supply chain, command prompt, cyber-attacks, code dependencies, hashes

Introduction

In the world's growing dependence on technology, cybersecurity has become extremely necessary due to the amount of daily life online. Digital assets, including banks' and private companies' access to personal information, can create a lot of damage if not secured properly, especially when used by billions of people. A large number of attacks and vulnerabilities are due to the insecurities in software supply chains which in-toto and TUF work to prevent.

Supply chains are the most basic framework for creating and distributing a product from start to finish. A traditional supply chain is the process of making and selling tangible items. For example, if a company makes laptops, they would follow the process illustrated in Figure 1. Firstly, they would gather materials needed to make a laptop like chips, screens, and keyboards. Then, they would bring these materials to a factory where the laptop would actually be put together. Subsequently, trucks, ships, and planes would transport these finished laptops to a store where people could browse and buy them.

Figure 1

Illustration of Traditional Supply Chain Structure

On the other hand, software supply chains are more abstract for both software developers and users; this enables attacks to occur more often within software supply chains. Steps in supply chains, specifically for software, are typically "chained" together meaning that if an attacker gets control of even a single step, they would have the ability to introduce vulnerabilities into the final product.

Taking a closer look at the software supply chain in Figure 2 reveals that the steps are very similar to what a traditional supply chain looks like. Starting out with code dependencies, software developers are then able to write and build their code. Afterward, their code is uploaded to repositories via the internet—instead of trucks, ships, and planes—where users can download the software from the cloud.

Figure 2

Illustration of Software Supply Chain Structure

Currently, many large software companies have insecurities in their software supply chains. Just last year, due to an insecurity in Apple's software supply chain, they were targeted in a $50 million ransomware hack of their supplier, Quanta (Mehrotra, 2021). Along with Apple, Microsoft and many other companies were hacked into by Alex Birsan through dependency confusion or namespace confusion attacks (Birsan, 2021). These occur in the beginning stage of the software supply chain by placing malicious code into an official public repository. Because Birsan named these adversary dependencies the same as the original dependencies, he was able to take down the largest technology companies in the world. There are also other ways an attacker can introduce these malicious threats within a chain. For example, if a hacker were to interpose something between two preexisting steps—like signing a bad version of a package—this malicious software could be added to the repository causing it to become vulnerable (Torres-Arias, 2019, p. 1394). Due to this, along with an accumulation of other international attacks, President Joseph Biden signed an executive order on May 12, 2021, titled

"Improving the Nation's Cybersecurity" (Executive Order 14028, n.d.). Section four specifically mentions the software supply chain and how more security revolving it is now required for these companies. In June 2022, Apple sent out a public service announcement concerning a new implementation of a cybersecurity capability they have been working on since these attacks occurred (Radcliffe, 2022).

Although there are currently many tools to secure each step of the software supply chain individually, in-toto is like no other framework because it works to secure the entirety of the software supply chain. Some point solutions for writing the code include using SVN, CVS, and git signing (Torres-Arias & Cloud Native Computing Foundation, 2020). In the build step, TPMs, HSMs, verifiable compilers, and reproducible builds are also good point solutions. For the package step, TLS, GPG, and TUF work well for security purposes. For a software developer, these are often too many to keep up with and track, so in-toto works to simplify this process.

However, implementation for in-toto has proven to be not so user-friendly. Many developers for Python-TUF find it difficult to implement in-toto due to the amount of documentation it takes to go through along with the debugging process. Therefore, in this study, I analyze how the Python-TUF in-toto proposal for usability concerns can be improved. I hypothesized that this could be accomplished by compiling all of the instructions in the verify section of the in-toto documentation and automating it. This automation is a relatively small part of the in-toto implementation for Python-TUF, but it is one of many first steps that will be used to create a smoother and more long-term process for Python-TUF developers' in-toto implementation.

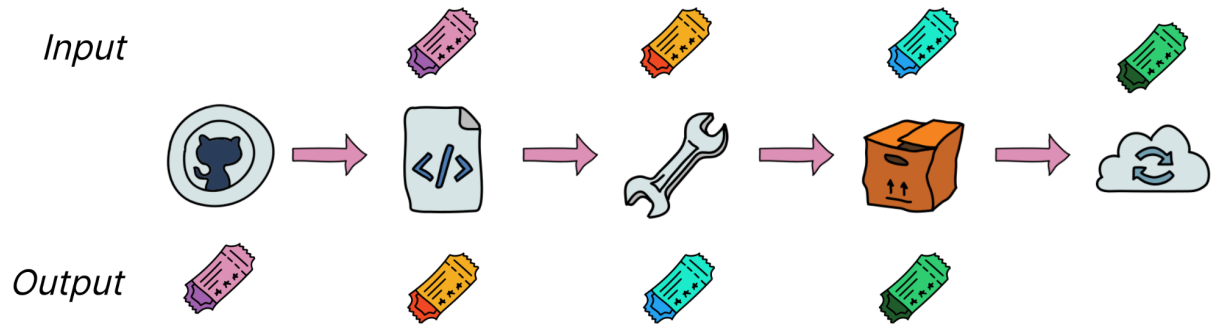
Literature Review

in-toto

in-toto is a framework that is used to secure the entirety of the software supply chain. Rather than having a single step checked by either git or TPMs, in-toto is able to create a very flexible and customizable framework that developers can mold to their liking. As shown in Figure 3, in-toto is able to secure each step by checking that the previous output is the same as the subsequent input. By applying this down the supply chain, developers can be sure that nothing malicious is being inserted into the process. It is important to note that this example is a very linear process compared to the reality of many software supply chains. Most software supply chains are not just one step after the other; instead, they go back and forth and even circulate between steps.

Figure 3

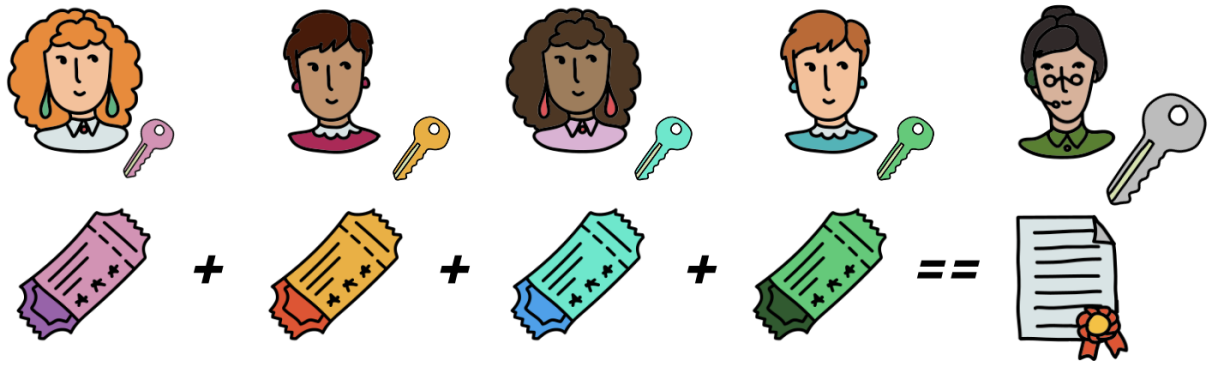
Illustration of Link Metadata



in-toto then has another layer of authentication called a layout depicted in Figure 4. This layout, formatted by the project owner, describes which users can sign off on what step and when.

Figure 4

Illustration of Link Metadata in Action and GPG Keys



In order to sign either link metadata or a layout, users need to use a GPG key that consists of an alphanumeric fingerprint. In Figure 4, each user has a key corresponding to their own link metadata that they need to sign. GPG keys are another way of protecting and securing this process. They are created by running the following line of code in a command prompt.

```
gpg --generate-key
```

The way GPG keys are secure is that each time the code to generate these keys are run, a public and private version is created. Referring to the left part of Figure 5, a public key can be recognized by all users. However, on the right side of Figure 5, the private version of that key can only be accessed by the user who created the key. When a developer signs their link metadata or layout, they can only sign with their private key, which will be displayed as the public key fingerprint for other users to recognize that the signature is valid.

The project owner's key is extremely important because it is the final piece of security on the software which is why it is larger in scale in Figure 4 compared to the other keys. This key is extremely important so it is often kept on a USB drive that is stored in a very secure place, most often a bank vault.

Figure 5

Illustration of Public and Private GPG Keys

Using all this information, in-toto is able to verify if each step is signed correctly and at the right time by following the layout. This is what specifically makes in-toto a framework. This framework structure is up to the project owner to decide and mold to their liking.

TUF

TUF, which stands for The Update Framework, is a framework for securing software update systems for developers. TUF works well with in-toto as the developers for in-toto are almost the same team that worked on TUF. TUF works to secure the "last mile" in the software supply chain while in-toto does everything up to that; TUF can be used to deliver updates and their corresponding in-toto metadata (Cloud Native Computing Foundation, 2022). TUF is written in a few different languages, but for this project, I worked closely with the Python version, Python-TUF.

Using a hierarchical organization, TUF works by creating four main different metadata roles: root, timestamp, snapshot, and targets. The first layer of authentication, also known as the

targets role, specifies what are the actual files that the clients want to download. This can include the software update that a user is trying to obtain. The metadata file for the targets role consists of hashes and the sizes of the actual files. Next is the snapshot metadata which lists all the version numbers of the metadata files other than the timestamp metadata file. This ensures that all clients will see a consistent view of all the files on the repository, like a snapshot. The next role is the timestamp metadata which acts as the heartbeat for the security checks. The timestamp role is frequently downloaded; in most cases, this means as frequent as once a day. Lastly, the most important role is the root metadata. This metadata specifies the other top-level roles along with the minimum number of keys required to sign a certain role's metadata, otherwise known as the signature threshold.

Materials and Methods

This section proposes a study that will work on the implementation of in-toto, a framework to secure the integrity of software supply chains, in Python-TUF, a framework for securing software update systems for developers writing in Python. This is done by automating a part of the process using a Python script file that can be run within a developer's command prompt. Following this, the procedure for the study is outlined, and planned interpretations of the data are discussed.

TUF Release with in-toto Attestations

In order to create a new release for TUF, there are certain instructions that are listed within a Markdown file on TUF's GitHub repository for how to do it with in-toto (Lukas Pühringer, n.d.). It describes the prerequisites, specific steps like tag and build, the verification step, and future user verification implementations. This instruction manual is incredibly long and difficult to implement as a single developer using a command prompt. When I tried it out, I encountered numerous bugs. Whether it was due to the key I was signing with or my command prompt simply breaking from some of the commands I ran that were listed within the instructions, trying to release my own updated version took a lot of trial and error. Therefore, my project tackles this issue by compiling a part of the instructions, shown in Figure 6, into a single Python script file that can be run in one line of code.

Figure 6

Screenshot of the Verify Section of the Release with in-toto Instructions

Download CD build job attestations

```
# Workaround to glob download '{wheel, sdist}.*.link' files from release page
cd_keyid=$(wget -q -O - https://github.com/${github_repo}/releases/tag/v${version} | \
    grep -o "sdist.*.link" | head -1 | cut -d "." -f 2)

wget -P .in_toto https://github.com/${github_repo}/releases/download/v${version}/sdist.${cd_keyid}.link
wget -P .in_toto https://github.com/${github_repo}/releases/download/v${version}/wheel.${cd_keyid}.link
```

Verify 'tuf-${version}.tar.gz' against policies in 'sdist.layout'

```
mkdir empty && cp dist/tuf-${version}.tar.gz empty/ && cd empty
in-toto-verify \
  --link-dir ../.in_toto \
  --layout ../.in_toto/sdist.layout \
  --gpg ${verification_keys[@]} \
  --verbose
cd .. && rm -rf empty
```

Verify 'tuf-${version}-py3-none-any.whl' against policies in 'wheel.layout'

```
mkdir empty && cp dist/tuf-${version}-py3-none-any.whl empty/ && cd empty
in-toto-verify \
  --link-dir ../.in_toto \
  --layout ../.in_toto/wheel.layout \
  --gpg ${verification_keys[@]} \
  --verbose
cd .. && rm -rf empty
```

How I approached automating the lines of code in Figure 6 can be broken down into the following steps: input, copy, and verify data. The first step, inputting data, gathers data that is specific to the user that in-toto verification needs. As shown below, the first piece of data in-toto needs is `github_repo`, the GitHub repository name, which tells in-toto where the software it is trying to secure is located. Next, I prompt the user to enter the current version of TUF they are trying to release and save that in `version`. I do the same for `gpg_id`, where I get the user's public key which in-toto uses later on.

```
github_repo = "athenachernandez/tuf"
version = input("Version of TUF: ")
```

```
gpg_id = input("GPG key id for layout: ")
```

The last piece of data needed for the release verification is the CD key id, `cd_id`. This process takes a bit more code to extract because it is listed in a file on the GitHub repository that the user provided. First I use `github_repo` and `version` that was provided by the user to get to the file on GitHub that contains the CD key id. Then, I use the `requests` library to get the site link into text format which I can then search through for the specific key by using the `re` library. I do this by finding the metadata file that looks like `sdist.*.link`. The asterisk means that any text can go there; in our case, we want to find the `sdist.cd_id.link`. Because the `findall()` function returns all instances of the `pattern`, I index the first one—since they should all theoretically have the same CD key id—and splice off `sdist.` and `.link` so all I am left with is the CD key id which I save in the variable `cd_id`.

```
# Get cd key id from tagged release
site_url =f"https://github.com/{github_repo}/
            releases/tag/v{version}"
response = requests.get(site_url)
site_text = response.text                  # Translate site to text

pattern = "sdist.*.link"
# Find link file complete name
result = re.findall(pattern, site_text)
cd_id = result[0][6:-5]                    # Slice for only cd key id
```

Next, is the second step: copy data. Because I am going through important metadata files that I do not want to disturb, I am going to copy all the files into a temporary directory and run in-toto verify there. I use a `with` statement and the `tempfile` library to create a temporary folder where I can move all the link metadata and layout metadata from the `.in_toto` folder to.

There is one file besides that of the contents within the `.in_toto` folder that I need to

copy over called the link metadata from the GitHub release. This will be used to verify the link metadata that we created when we let in-toto run its verification process. Therefore, similarly to when I extracted the `cd_id` from the GitHub repository by getting the text version of a GitHub link, this time I used the `cd_id` to get the actual contents of the link metadata. Then, I make a new file within the temporary directory, and write the contents to that file.

```
# Creates temporary folder
with tempfile.TemporaryDirectory() as temp_dir:
    # Link metadata from GitHub release
    site_url = f"https://github.com/{github_repo}/
                releases/download/v{version}/sdist.{cd_id}.link"
    response = requests.get(site_url)
    site_text = response.text                # Translate site to text

    # Make new file in temporary directory
    release_metadata_file = open(f"{temp_dir}/sdist.{cd_id}.link",
                                                              "w")
    release_metadata_file.write(site_text)
    release_metadata_file.close()
```

Now, I assume the Python script file I am writing will be sitting within the user's main folder for the TUF project. That way, when I use the `os` library to get the current working directory, I can go directly into the `.in_toto` folder by the `source_dir` path which I have on the next line. The assumption is important because if my Python script file is anywhere else, this code will not do its job at all. I use the `os` library again to retrieve all the contents that are in the `.in_toto` folder and save it in a list called `in_toto_contents`. Then, I loop through the contents and use the `shutil` library's `copyfile()` function to copy all the files over to the temporary directory one by one.

```
    # Still in the with statement
    cwd = os.getcwd()                        # Gets current directory
```

```
source_dir = f"{cwd}/.in_toto/"
in_toto_contents = os.listdir(f"{cwd}/.in_toto/")
# Copy .in_toto dir to temp folder
for item in in_toto_contents:
    src_file = source_dir + item         # Source file path
    dst_file = f"{temp_dir}/{item}"      # Destination file path
    shutil.copyfile(src_file, dst_file)  # Copying each file
```

Now, I have reached step three: verify data. But before I can do that, I need to import a few `in-toto` libraries. These are to format the data I have thus far in ways that the `in_toto_verify()` function's parameters agree with.

```
from in_toto.models.metadata import Metablock
from in_toto.verifylib import in_toto_verify
from securesystemslib.gpg import functions as gpg_interface
```

I use `Metablock` to load the layout itself. The `os.path.join()` is just to combine the temporary directory's path with `sdist.layout` to create one longer path where the layout is located. Then, the `layout_key_dict` is a dictionary, a unique Python data structure, where I can store all the keys needed. As I mentioned previously, the basic example of how in-toto works that I provided is a very linear example. In reality, it is not just one user signing off one step, it is many users, and this dictionary is where these keys can be stored to input to the `in_toto_verify()` function. The last parameter that `in_toto_verify()` takes in is the path to the folder where all the metadata is stored. Since I copied all the data into the temporary folder—which is still open because I am still indented in the `with` statement where the folder was created—I can give it `temp_dir`.

```
# Still in the with statement
layout = Metablock.load(os.path.join(temp_dir, "sdist.layout"))
layout_key_dict = {}        # Dictionary to store keys
```

```
layout_key_dict.update(gpg_interface.export_pubkeys([gpg_id]))
in_toto_verify(layout, layout_key_dict, link_dir_path=temp_dir)
```

This completes the Python script file that can be visited in a GitHub gist that is currently undergoing revisions to be implemented within the TUF repository soon (Hernandez, 2022).

Results

Now instead of having to work through the commands provided in Figure 6, all a Python-TUF software developer would need to do is run the following in their command prompt.

```
python3 verification.py
```

To review, I simply automated a small part of the verification instructions within in-toto's release for Python-TUF by structuring a Python script file into three parts: input, copy, and verify data. The command prompt should return either "verification successful" or "build artifacts do not match." These are the same responses that occur when successfully executing all the commands within Figure 6. If the command prompt returns "verification successful," the release with in-toto was successful because the link and layout metadata matched up with in-toto's requirements. If the command prompt returns that the "build artifacts do not match," then this means the link and layout metadata did not match up and in-toto thinks there might be malware within the new software that is trying to be updated.

Discussion

The data supports my original hypothesis that the usability concerns in the in-toto proposal for Python-TUF could be improved. I did this by automating part of the release with in-toto instructions within Python-TUF by compiling all of the command prompts in the verify section. Now, Python-TUF developers do not need to struggle with the command prompt when trying to release a new version of the software. There is not much current research for simplifying the user implementation specifically for the Python-TUF in-toto crossover, but my mentors mentioned that anything that is able to automate the current instructions is better than its current state. However, this code only works for one type of metadata file, `sdist`, as I have not implemented the `wheel` version. This is a very simple future implementation that can be added

once the logistics are figured out for my current Python script that my mentors are reviewing. There were a few roadblocks throughout my research that slowed down the process. First of which was using a Python virtual environment. In the link metadata, I had to exclude the Python environment packages that were being recorded because if not then my build artifacts would not match. Another issue was the naming of my PyPI project. Although it was more of a formality, it disrupted my code from releasing properly in my GitHub repository which slowed down the verification process.

Conclusion

Overall, in a world becoming more dependent upon technology, software supply chain security is an area of research that needs to be improved and reanalyzed with more precision. Most software developers are deterred from using software supply chain security due to the hassle it creates because of the overwhelming amount of softwares they have to use to secure the chain. in-toto is working on solving this, however, its implementation is quite rigorous for many developers. Therefore, in this study, I worked to automate a small part of the process by compiling many command prompt instructions into one Python script file that can be run in one line of code. That way, usability concerns are improved in the Python-TUF repository. In the future, I hope to expand this automation to other parts of the in-toto proposal for TUF possibly through the use of more script files.

Acknowledgements

This research opportunity would not have been possible without my mentors Justin Cappos, Lukas Pühringer, and Aditya Sirish at the Center for Cybersecurity and Secure Systems Lab at New York University's Tandon School of Engineering. I would also like to extend my sincere thanks to Aram-Alexandre Pooladian for his mentorship in the Data Science course which would not have been possible without the Winston Foundation. I would be remiss in not mentioning and thanking Catherine Tissot, Victoria Zhang, Shreya Desai, Matthew Leingang, all other course assistants, and my fellow peers from the GSTEM program within New York University's Courant Institute of Mathematical Sciences.

References

Birsan, A. (2021, February 9). Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. *Medium*. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

Cloud Native Computing Foundation. (2022). *The Update Framework* (Version 1.1.0) [Computer software]. https://theupdateframework.io/

Exec. Order No. 14028, 3 C.F.R. (). https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity

Hernandez, A. (2022, August 10). *verification.py*. GitHub Gist. Retrieved August 15, 2022, from https://gist.github.com/athenachernandez/4369cd5e63f056dfea62eef322528960

Icons [Image]. (n.d.). Icons8. https://icons8.com/

in-toto. (n.d.). *in-toto* (Version 1.2.0) [Computer software]. https://in-toto.io/

Lukas Pühringer. (n.d.). *Release with in-toto attestations*. GitHub. Retrieved August 15, 2022, from https://github.com/lukpueh/tuf/blob/017031b761fe08e6e037c89fad7e6c42ab5d1eb0/docs/RELEASE_with_in-toto.md

Mehrotra, K. (2021, April 21). Apple Targeted in $50 Million Ransomware Hack of Supplier Quanta. *Bloomberg*. https://www.bloomberg.com/news/articles/2021-04-21/apple-targeted-in-50-million-ransomware-hack-of-supplier-quanta#xj4y7vzkg

Radcliffe, S. (2022, July 6). *Apple expands industry-leading commitment to protect users from highly targeted mercenary spyware*. Apple Newsroom. Retrieved August 15, 2022, from

https://www.apple.com/newsroom/2022/07/apple-expands-commitment-to-protect-users-from-mercenary-spyware/

Torres-Arias, S. (2019). in-toto: Providing farm-to-table guarantees for bits and bytes. *Proceedings of the 28th USENIX Security Symposium*. https://www.usenix.org/system/files/sec19-torres-arias.pdf

Torres-Arias, S., & Cloud Native Computing Foundation. (2020, September 4). *in-toto: Securing the Entire Software Supply Chain* [Lecture video]. YouTube. https://www.youtube.com/watch?v=W-5io6v3S1Y