Athens University of Economics and Business, Dept. of Informatics
Master Thesis in Information Systems: Development and Security

Artifact Attestations

Georgiadis Eleftherios

ID f3312304

Supervisor

Prof. Xylomenos Georgios

Athens University of Economics and Business

Co-supervisor

Fotiou Nikos

Athens University of Economics and Business

Academic Year 2024/2025

Additional resources

This thesis is based on a project hosted on GitHub.

This project is a monorepo (inludes infrastructure files) for three microservices, that implement a very simple rest api about a car rental service.

The repo features pipelines for GH Actions to demonstrate artifact attestations.

Link to GitHub repository:

lefosg/quarkus-rest-car-rental-service

Security is only as useful, as usable

Abstract

In today’s tech setting, the cloud has revolutionized the way we develop, build, deploy and monitor our applications. Businesses "lift and shift" their infrastructure to the cloud where their applications are hosted. Because of that, the process of going from code to executable has somewhat changed. This transition has a name, and that is Development and Operations (DevOps). In DevOps, we have automated Continuous Integration/Continuous Delivery (CI/CD) pipelines that build, test and deploy our code with minimal manual effort required.

Though DevOps may be the most substantial part for some people, the attack surface of the CI/CD pipelines is large enough. So, nowadays, the trend is about Development, Security, and Operations (DevSecOps), which includes the security aspect of DevOps as well. There is a plethora of attack vectors in the Software Supply Chain, such as introducing malicious software in a dependency, compromising the source code repository, and many more. So one question that has raised, is, how can I know if the software I am currently running, is indeed the one that was intended to be tested, built or distributed (just to name a few steps in the chain)?

Artifact Attestations are a way of providing some confidence to the consumer of an artifact, about some claims made for it. For example, to which commit does this artifact correspond to, which build system built it, or, was the code peer reviewed? This is done by providing an extra piece of information to the consumer, an *attestation*. There are many types of attestation, such as *provenance attestation* which provides proof about the build process of an artifact, the *SPDX attestation* (or *SBOM attestation*) and many more.

In this thesis we explore how this is technically feasible through a CI/CD platform, *GitHub Actions*, what are the security benefits, and limitations of using artifact attestations. We will also dive into other technologies and standards involved in the process such as Sigstore, Supply-chain Levels for Software Artifacts (SLSA), in-toto, and others.

Περίληψη

Στο σημερινό τεχνολογικό περιβάλλον, το υπολογιστικό νέφος (cloud) έχει φέρει επανάσταση στον τρόπο που αναπτύσσουμε, κατασκευάζουμε, διανέμουμε και παρακολουθούμε τις εφαρμογές μας. Οι επιχειρήσεις μεταφέρουν ("lift and shift") τις υποδομές τους στο cloud, όπου φιλοξενούνται οι εφαρμογές τους. Εξαιτίας αυτού, η διαδικασία διανομής λογισμικού ξεκινώντας από τον κώδικα μέχρι το τελικό εκτελέσιμο έχει αλλάξει σημαντικά. Αυτή η μετάβαση έχει όνομα, και αυτό είναι Ανάπτυξη και Λειτουργία (Development and Operations - DevOps). Στο DevOps, έχουμε αυτοματοποιημένες διαδικασίες (pipelines) Συνεχούς Ενοποίησης/Συνεχούς Διανομής (Continuous Integration/Continuous Delivery - CI/CD) που κατασκευάζουν, δοκιμάζουν και διανέμουν τον κώδικά μας αυτοματοποιημένα, με ελάχιστη απαίτηση για χειροκίνητη παρέμβαση.

Αν και για τους περισσότερους το DevOps μπορεί να είναι το πιο ουσιαστικό κομμάτι, η επιφάνεια επίθεσης (attack surface) των CI/CD pipelines είναι αρκετά μεγάλη. Έτσι πλέον, η τάση αφορά την Ανάπτυξη, Ασφάλεια και Λειτουργία (Development, Security, and Operations - DevSecOps), το οποίο ενσωματώνει και την ασφάλεια στο DevOps. Υπάρχει πληθώρα επιθέσεων στην Αλυσίδα Λογισμικού (Software Supply Chain), όπως η εισαγωγή κακόβουλου λογισμικού σε μια εξάρτηση (dependency), η παραβίαση του αποθετηρίου πηγαίου κώδικα (source code repository) και πολλά άλλα. Έτσι, ένα ερώτημα που έχει προκύψει είναι το εξής: πώς μπορώ να γνωρίζω ότι το λογισμικό που εκτελώ αυτή τη στιγμή είναι πράγματι αυτό που προοριζόταν να αναπτυχθεί, δοκιμαστεί, να κατασκευαστεί ή να διανεμηθεί (απλώς για να αναφέρουμε μερικά βήματα της διαδικασίας);

Οι Βεβαιώσεις Τεχνουργημάτων (Artifact Attestations) αποτελούν έναν τρόπο παροχής εμπιστοσύνης στον καταναλωτή ενός artifact σχετικά με ορισμένους ισχυρισμούς που γίνονται γι' αυτό. Για παράδειγμα, σε ποιο commit αντιστοιχεί το artifact, ποιο σύστημα κατασκευής (build system) το δημιούργησε, ή αν ο κώδικας υποβλήθηκε σε έλεγχο από τρίτους (peer review). Αυτό επιτυγχάνεται μέσω της παροχής μιας επιπλέον πληροφορίας στον καταναλωτή, μιας βεβαίωσης (attestation). Υπάρχουν πολλοί τύποι βεβαιώσεων, όπως η βεβαίωση προέλευσης (provenance attestation), που παρέχει αποδείξεις για τη διαδικασία κατασκευής ενός τεχνουργήματος, η βεβαίωση SPDX (ή SBOM attestation) και πολλές άλλες.

Στην παρούσα διπλωματική εργασία, διερευνούμε πώς αυτό είναι τεχνικά εφικτό μέσω μιας CI/CD πλατφόρμας, την GitHub Actions, ποια είναι τα οφέλη ασφαλείας, καθώς και οι περιορισμοί της χρήσης των artifact attestations. Επιπλέον, θα εξετάσουμε και άλλες τεχνολογίες και πρότυπα που εμπλέκονται στη διαδικασία, όπως το Sigstore, τα Supply-chain Levels for Software Artifacts (SLSA), το in-toto και άλλα.

Acknowledgments

For the completion of the present thesis, I want to thank my supervisors, Prof. Xylomenos Georgios and Fotiou Nikos for their help and guidance throughout the process.

Also, I would like to thank my professors Mr. Giakoymakis and Mr. Zafiris who handed us the microservice project in the second semester of my master, which is the use case of my thesis.

Finally, I want to thank my colleague Gregory who inspired me to setup my own home lab and enhance my DevSecOps skills.

Contents

1 Introduction

1.1 Motivation and Problem Statement

In mundane computer use, many of us "average users" may find ourselves in a situation where we have wondered: "How am I sure that this piece of software I am currently running, is secure?", or "Am I being exposed to any risks by using this software?". And that, is indeed a common thought, since most of the times we are software consumers, not producers/developers. On the other hand, software engineers can similarly ask "how do I know that the software I develop does not contain any malicious code?".

The process of going from source code to final product is well defined. Typically, a developer would manually build the source code locally using a build tool (e.g., Maven) and upload the final software product in a repository/distribution platform (e.g., an organization's webpage, DockerHub, PyPI etc.). However, due to the increase of DevOps' popularity, a transition has been made from manually executing the aformentioned steps, to creating CI/CD pipelines which automate the build, test, package, delivery, deployment and monitoring process, speeding up the Software Development Life Cycle (SDLC) as well. A build platform is used to run these pipelines (e.g., Jenkins, GitHub Actions etc.), which can be either self-hosted, or run on the cloud.

So now, creating the final software product is a well established procedure. Anyone can view the pipeline - which is usually a simple script - and consider its security aspects. But, the attack surface is just too vast. For example:

- A developer himself may be the adversary and insert malicious code

- A third party dependency imported into the source code may be malicious
- The build platform may be compromised
- The package registry used to upload the final product is compromised

The majority of these attack vectors fall under the category of Software Supply Chain Security. Most of the attacks in Software Supply Chain aim in the injection of malicious code into the software product [10]. A popular example is that of SolarWinds, a tech company which provides tools for ICT infrastructure monitoring (i.e., highly privileged software). Their build system was compromised, leading to the injection of malicious code in every new build of their software product. When the product was distributed, it contained the malware introduced by their builder, so customers who downloaded the specific versions, had their systems infected with that malware.

It may be impossible to ultimately detect and prevent in real-time or on-time such attacks, but there is a new method which helps us mitigate vulnerabilities of the supply chain. Artifact Attestations provide some confidence about the authenticity and integrity of an artifact. Specifically, provenance attestations, are attestations which are created during the build process (amidst the pipeline). Their purpose is to prove that an artifact was indeed produced by a specific builder[14]. Attestations are verifiable too, and must be verified, otherwise their security purpose is defeated. There are multiple underlying technologies that help make artifact attestations a secure means of providing trust to the downstream consumers of an artifact.

In this thesis we explore how artifact attestations are technically feasible by creating a CI/CD pipeline and running it on GitHub Actions. The use case is about a Kubernetes cluster which hosts three microservices (a REST API about a car rental service) and how attestations secure the process of deploying the microservices in the cluster. We consider the levels of trust they provide, and how an adversary can overcome the security benefits they provide.

1.2 Thesis Structure

Chapter 2
We start by referencing and analyzing background knowledge and technologies required. Definitions for core concepts are provided.

Chapter 3

In this chapter we discuss the concept of DevSecOps. What is it, why does it fit so well in the current cloud based model and why security matters in its case.

Chapter 4
In this chapter we showcase Sigstore, a universal artifact signing system. The system design is explained, as well as its security benefits and limitations. Basic knowledge of Sigstore is required, because GitHub Actions platform leverages it to sign attestations (more on that on chapter 6.

Chapter 5
In this chapter we explore the software supply chain. What is it, and what are the attack vectors it introduces.

Chapter 6
In this chapter we focus on artifact attestations, and how all the previous chapters are connected. The use case is analyzed, meaning, the setup of the CI/CD pipeline in GitHub Actions, the Kubernetes cluster architecture and how artifact attestations fit in there.

Chapter 8
Finally, the thesis is concluded by expressing thoughts on artifact attestations.

2 Background

In this section, we define core concepts required to understand artifact attestations, DevSecOps, and Sigstore. The purpose is to build a holistic view of the problem.

2.1 Software Development Life Cycle

Software is a "thing". It is something that "runs" on a machine, and makes our life easier. Though software may not be "alive", there is term called Software Development Life Cycle. How is software produced and consumed, from start to end. From text which producers (developers) call *code*, to 0s and 1s which end users call *file*, *executable* and more. A general overview of the SDLC can be seen in the following picture:

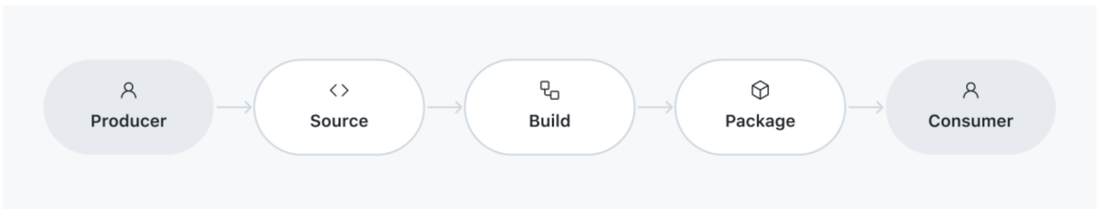


Figure 2.1: Software Development Life Cycle Overview (source: Where does your software (really) come from? [15])

What matters the most to understand right now, is that software is not created and distributed only **once in its lifetime**, but it needs to be **maintained**. And that is an expensive procedure. It takes a lot of effort, man-hours and money to efficiently maintain software.

Let's give a practical example of that. A developer is writing a Java application on his computer, say, a calculator app for Windows, using Maven as a build tool. The typical procedure would be:

1. Write the Java code
2. Write tests
3. Test, build and package using *mvn package*
4. Use a tool to convert the .jar file to .exe
5. Upload the .exe file to a distribution platform (e.g., SourceForge)

As more features are installed, this procedure needs to be repeated again and again, manually, on the developer's end. A solution for that is automated build systems (like the one in our use case, GitHub Actions), which automate this process by running those commands after an event has been triggered. This will be analyzed later on.

2.2 OpenID Connect

Have you ever seen the "Log in or register with Google/Microsoft/Facebook" button in a login or registration page? Chances are that the underlying mechanism of it is OpenID Connect (OIDC). OpenID Connect is a delegated authentication protocol [3]. It enables services to authenticate to each other while handling the identity management part of it as well. OIDC is the fundamental authentication mechanism of Sigstore, that's why an explanation of it is done.

Suppose we have developed an application, and we want to have our clients register to it, and log in later on. Storing and managing credentials is a risk. In the event of a data breach, if the clients' credentials are not securely stored in the database, the result would be catastrophic both for the clients and the business running the application. And even before that, there has to be a member in the team with the expertise and knowledge to properly implement the best practices for storing the credentials. So, instead of maintaining an in-house authentication scheme, OIDC can be leveraged as an alternative. If a client can be successfully authenticated by an Identity Provider (IdP) like Google or Microsoft, then we consider this user authenticated to our application as well.

An OIDC flow results in the Relying Party (RP) - that is, our app's backend - receiving a token. That token is called *ID Token*. It contains metadata about the authentication process (e.g., nonces, urls), and personal attributes about the

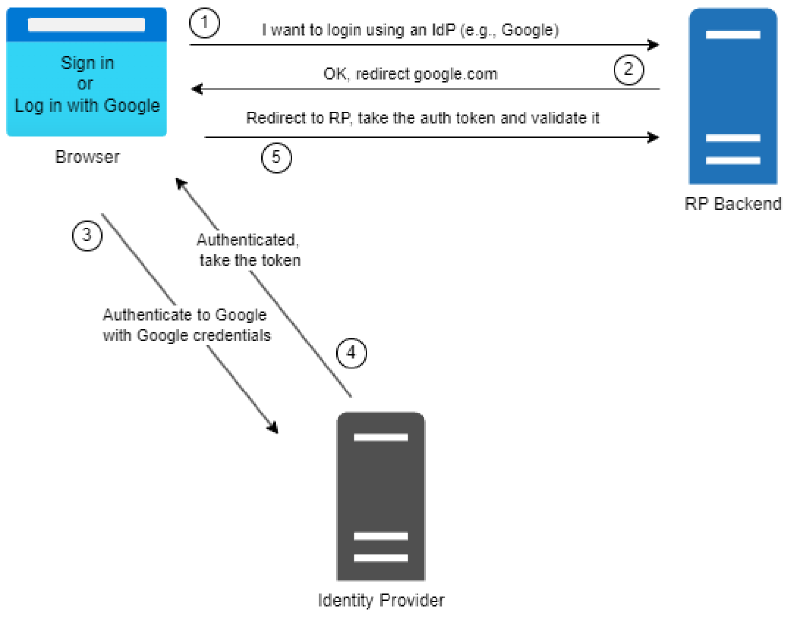


Figure 2.2: OIDC flow simplified

subject authenticated. If a subject gets authenticated with say with Google, to an OIDC-enabled application, then the application would like to know some information about that specific subject like email address, name or phone number. These attributes are included in the ID Token, and are called *claims*. The token itself is a JSON Web Token (JWT), and is signed by the IdP. That implies that it must be verified by the RP, otherwise its security purpose fails.

The last step (5) of the picture above can be slightly different depending on the case, and the way a developer implemented the OIDC flow. The token can be sent directly to the RP through back-channel (server to server communication), meaning that the user agent i.e., the browser, will not be an intermediate. Generally, that is thought to be a secure means of sending the token. Another way is to send a code instead of the token, and then the RP will exchange that code for a token.

2.3 What is an Artifact

There are quite a few definitions in literature for the term *artifact*:

ISO/IEC 19506:2012 [6]:

"An artifact is a tangible machine-readable document created during software development. Examples are requirement specification documents, design documents, source code and executables."

SLSA Standard [11]:

"An immutable blob of data; primarily refers to software, but SLSA can be used for any artifact e.g. a file, a git commit, a directory of files (serialized in some way), a container image, a firmware image."

GitHub [15, 17]:

"That final stage of metamorphosis, that something else that source code becomes, is what we usually refer to as a "software artifact,"and after their creation artifacts tend to spend a good chunk of time at rest, waiting to be used" or, "a file or collection of files produced during a workflow run."

All these definitions are slightly different, but identical in their core. Any piece of bits and bytes can be more or less be called an *artifact*. From now on we will rely on the term *artifact* instead of *software product*, *executable* or any similar terms.

2.4 Attestations

An attestation is a signed document (e.g., JSON) which contains claims/metadata about a subject/artifact, created by an entity that can be verified [15]. It is an *assertion* about a set of facts regarding an *artifact*. Why would there be a need for artifact attestations? Because we want to verify it. Why verify? To gain trust.

2.4.a Attestations vs Signatures

A signature is the encrypted hash of an artifact. When a signature is verified, the *context* transferred from producer to consumer is that this artifact's integrity and authenticity is valid. However, what this artifact is, how it is described and other properties it may have are not *visible* through the signature verification. This communication gap between producer and consumer is filled in by attestations. Now, a producer can export a set of claims about an artifact and sign them. If the properties have been tampered for some reason, then the signature verification would fail.

GitHub now supports artifact attestations through GitHub Actions. There are premade actions in the marketplace which implement exactly this functionality i.e., create artifact attestations. Sigstore is used as the underlying signing mechanism.

2.5 Dead Simple Signing Envelope (DSSE)

The Dead Simple Signing Envelope (DSSE) is a data format which is used to sign arbitrary data, not just JSON. It overcomes some problems which existing solutions have such as JSON Web Signature (JWS) which has weak implementations, or canonical JSON which poses a large attack surface [16]. Below we can see the body of a DSSE:

```json
{
  "payload": "<Base64(SERIALIZED_BODY)>",
  "payloadType": "<PAYLOAD_TYPE>",
  "signatures": [{
    "keyid": "<KEYID>",
    "sig": "<Base64(SIGNATURE)>"
  }]
}
```

The fields *payload* and *signatures* are pretty much self explanatory. For *payloadType*, it can either be a URI which should resolve to a human readable description e.g. https://example.com/MyMessage/v1-json or MediaType e.g. application/vnd.in-toto+json. In the last case, the MediaType should be an implementation specific schema.

DSSE is used by in-toto and Sigstore, which will be explained on later in the thesis.

3 DevSecOps

In this chapter we will discuss the notion of DevOps and DevSecOps. A definition will be given, some practical examples of how DevOps influences engineers, and why security matters within this context.

3.1 What is DevOps

We will not focus too much on what DevOps is in theory, but since artifact attestations are used in this context, we will briefly explain it.

Typically, a developer would write the code for an application, build and test it, and call it a day. Then the *operations* team would take over, and deploy the application. If any troubleshooting had to be done, the *development* and *operations* teams would have to communicate in order to resolve the issue ("it works on my machine", but not in a deployment environment). That's where the methodology of DevOps comes into rescue. This methodology promotes the tight communication and collaboration of the *development* and *operations* teams - thus, *DevOps* [7].

While, in theory, DevOps does not focus on tools and implementations, a standardized way to achieve the aforementioned goal, is automation. Through automation, we build CI/CD pipelines which speed up the software delivery process, from source (i.e., what *devs* see) to deployment (i.e., what *ops* see) while this comes especially handy in a cloud setting, where frequent releases are rolled out [7].

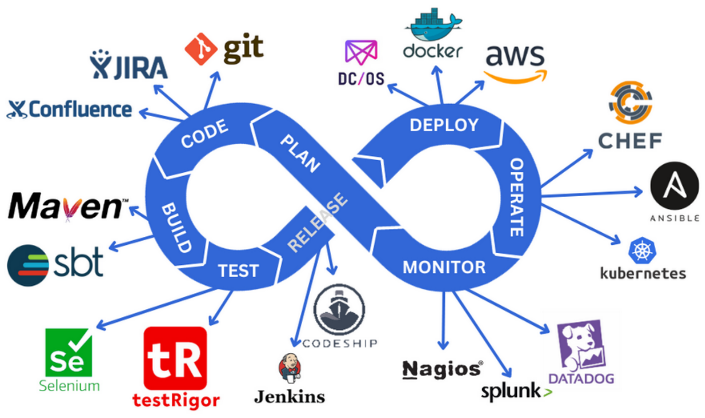


Figure 3.1: CI/CD steps and tools (source: What is CI/CD? [5])

3.2 CI/CD pipelines

CI/CD stands for Continuous Integration and Continuous Delivery, and is a core concept of the DevOps methodology. CI/CD pipelines are what make the process of building and distributing software products possible in an automated manner. "By automating the process, the objective is to **minimize human error** and **maintain a consistent process** for how software is released" as said by RedHat [13]. Also, automation provides security benefits and confidence that the result is as predictable as possible, while speeding up the SDLC. Assuming that code is version controlled with a system like *git*, for a specific commit, every developer should be able to execute the exact same pipeline and produce identical results in each run.

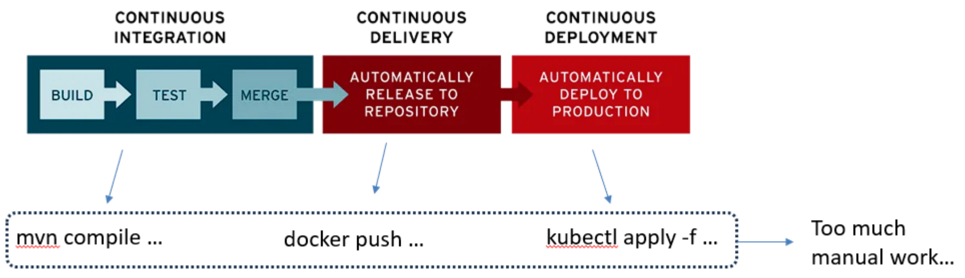


Figure 3.2: Manual to automated

Continuous Integration (CI) refers to the building process of the software product. That's the "metamorphosis" of source code to a usable piece of software e.g., a .jar file, a container image etc. That includes setting up the required runtime tools, compiling, testing and create the release of the software product.

Continuous Delivery (CD) (or sometimes called Continuous Deployment), is about taking the output(s) of the CI process - meaning the software product - and distributing it or deploying it straight to a dev/qa/production environment. For example, a container image may have to be deployed to a Kubernetes cluster right after it was built, because, a new feature for the corresponding service was created. The following image showcases such a scenario:

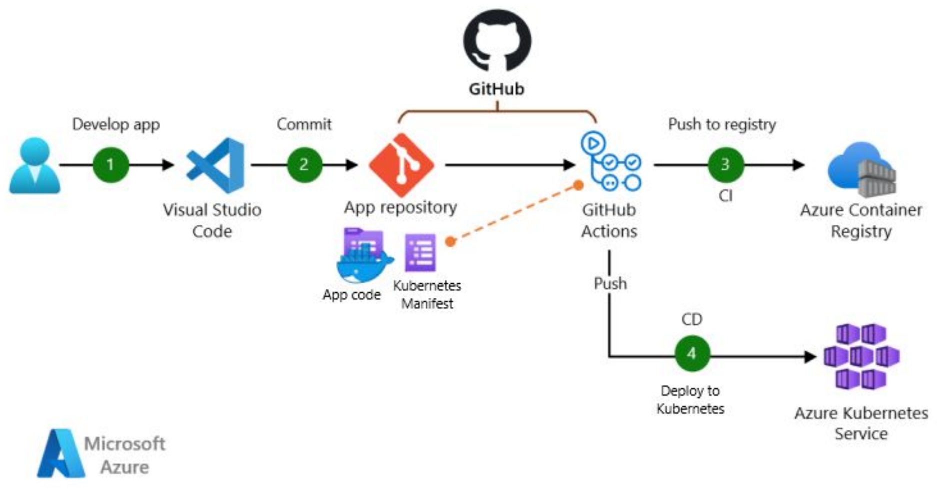


Figure 3.3: Sample CI/CD sketch (source: Apply software engineering systems [8])

3.2.a CI/CD in practice (example)

As discussed, the scope is to automate this procedure. We will better understand how this happens by diving into the technicalities. We use GitHub Actions, a CI/CD platform for executing such pipelines. Once a developer pushes a commit, a workflow starts running. Let *ci.yaml* be the following sample workflow file, which builds a container image from Java source code, using Maven:

```
name: Build Image

on:
  push:
    branches: [ "main" ]

```

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: '17'

      - name: Maven Package
        run: mvn clean package

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3.7.1

      - name: Log into registry
        uses: docker/login-action@v3.3.0
        with:
          username: ${{ vars.DOCKERHUB_USERNAME  }}
          password: ${{ secrets.DOCKERHUB_TOKEN  }}

       - name: Build and push Docker image
         uses: docker/build-push-action@v6.9.0
         with:
           tags: myaccount/myimage:latest
           file: ./Dockerfile
           push: true
```

This is a YAML file which defines the steps for a CI/CD pipeline for GitHub Actions. Upon receiving a *push* event on branch *main*, GitHub spawns a runner - a fresh Virtual Machine (VM) - and executes each step of the job *build* sequentially. The author of the pipeline declares which actions should run. Those actions are mainly written by other developers (there is an actions marketplace), so we choose from there which fit our needs, with the *uses* keyword. The steps followed are pretty much self explanatory:

1. The repository is cloned locally in the runner (actions/checkout action)

2. Java 17 is installed in the runner (actions/setup-java)
3. Maven is executed (it was installed in the previous step along with Java) → the .jar is produced and persisted temporarily in the runner
4. Docker buildx is set up (docker/setup-buildx-action)
5. Log in to DockerHub registry (docker/login-action)
6. A container image named *myaccount/myimage:latest* is created (docker/build-push-action) → because *push* equals *true*, the image is sent to a container registry as well (DockerHub by default, that's why there is a need for a login step)

Because this job builds and pushes a container image to a registry, we can say that it does the Continuous Integration and Delivery parts of the pipeline.

Similarly, there can be another workflow which defines the deployment of this container image. The following example (let *cd.yaml*) pulls the image built from the previous workflow, and pushes it to an Azure Kubernetes Service (AKS) cluster:

```yaml
name: Deploy Image To AKS (Azure Kubernetes Service)

on:
  workflow_run: Build Image
  types:
    - completed

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Log into registry
        uses: docker/login-action@v3.3.0
        with:
          username: ${{ vars.DOCKERHUB_USERNAME  }}
          password: ${{ secrets.DOCKERHUB_TOKEN  }}

       - name: Azure login
        uses: azure/login@v1.4.6
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}

```

```
         - name: Set up kubelogin for non-interactive login
           uses: azure/use-kubelogin@v1
           with:
             kubelogin-version: 'v0.0.25'

        - name: Get K8s context
          uses: azure/aks-set-context@v3
          with:
            resource-group: ${{ env.RESOURCE_GROUP }}
            cluster-name: ${{ env.CLUSTER_NAME }}

        - name: Deploys application
          uses: Azure/k8s-deploy@v4
          with:
            action: deploy
            manifests: ./k8s-manifests/mypod.yaml
            images: myaccount/myimage:latest
```

Again, the steps are as simple as they can get:

1. Log into the DockerHub registry
2. Login to Azure
3. Set up kubelogin (to authenticate the *kubectl* client to AKS)
4. Get the K8S context in order to configure *kubectl* client to point to the AKS K8s instance
5. Finally, deploy the image *myaccount/myimage:latest* using the manifest file *k8s-manifests/mypod.yaml*

With minimal effort we have defined a way to automatically build, test, containerize, and deploy our application. Now, a ground truth for developers, maintainers and stakeholders of a repository, are a set of CI/CD pipelines defined for that repo.

3.2.b Workload Identities

At this point, it is important to note that the *entity* that executes the pipeline is a *workload identity* i.e., the GitHub runner. A workload identity, is an identity assigned to a *software workload* (e.g., container, VM, script) [22]. This is needed because the running software needs to authenticate other services. As a result, this entity can be authenticated, and thus get authorized to access external resources. The GitHub runner that runs the pipeline provided in the example above has a workload identity, and it is used to authenticate to services such as Azure, DockerHub etc.

There are multiple security concerns about workload identities. In order to authenticate to an external service it - most likely - needs to provide user/password credentials. Such cases pose a security issue. Think about a scenario where malicious code was injected somehow in the pipeline, and its purpose it to steal information stored locally in the runner, and sends it to an unknown IP address. In reality, our build system is compromised. Is sensitive information stored in the software? How is it stored? GitHub addresses those issues by using secrets, OpenID Connect, and some other security hardening tips[1].

3.3 Security in DevOps (DevSecOps)

Traditionally, security requires **manual** effort, meaning costly man-hours. For instance:

- Security and quality code reviews (CVEs, CWEs, best practises)
- Theat Modeling (TM)
- Static Application Security Testing (SAST)
- Dynamic Application Security Testing (DAST)
- Software Composition Analysis (SCA)
- Dependency checking
- Secure repository rules (e.g., branch security)

and many more, are procedures required to assure the security posture of an application.

However, do these procedures fit in the context of DevOps? **DevSecOps** introduces the notion of automating security procedures, as part of a pipeline. There are quite some challenges and limitations in that field. Can all security aspects be automated? Can they even be fully automated? If not, then at what degree? For an organization, it is important to maintain the rapid software deployment procedure - which is practically their functional requirement - while at the same

[1]https://docs.github.com/en/actions/security-for-github-actions/security-guides/security-hardening-for-github-actions

time assuring the security quality of their product [12]. Such work exists. For example, Van Landuyt et. al [21] have studied if, and, in what extent can threat modeling get automated within the context of DevSecOps by examining available tools. So, it seems that DevSecOps engineers are somewhat dependent on the tools that exist. Meaning that producers of the security tools are required to extend the capabilities of their products, to fit the needs of their DevSecOps-enabled customers.

In our use case, GitHub Actions provides predefined actions available on the marketplace for security purposes. One of them is about **artifact attestations**, and more specifically provenance attestation which will be discussed in detail in chapter 6. Other actions may include other security measures, such as SAST, container vulnerability scanning etc. Below, an example yaml script is provided to demonstrate how DevSecOps could potentially work with GitHub Actions:

```yaml
name: Build Image

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: '17'

      - name: Build with Maven and Analyze with Sonarqube
        env:
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
          SONAR_HOST_URL: ${{ secrets.SONAR_HOST_URL }}
        run: mvn -B --file ./pom.xml verify
          org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
          -Dsonar.projectKey=myproject
```

```
            -Dsonar.projectName='myproject'

        - name: Set up Docker Buildx
          uses: docker/setup-buildx-action@v3.7.1

        - name: Log into registry
          uses: docker/login-action@v3.3.0
          with:
            username: ${{ vars.DOCKERHUB_USERNAME  }}
            password: ${{ secrets.DOCKERHUB_TOKEN  }}

         - name: Build and push Docker image
           uses: docker/build-push-action@v6.9.0
           with:
             tags: myaccount/myimage:latest
             file: ./Dockerfile
             push: true

        - name: Run Trivy vulnerability scanner
          uses: aquasecurity/trivy-action@0.28.0
          with:
            image-ref: myaccount/myimage:latest
```

The steps are the same as with subsection 3.2.a, but two security measures have been added.

1. One is scanning with Sonarqube [2] and sending the results to a Sonarqube server (variable *SONAR_HOST_URL*)
2. Second is scanning the container image produced for vulnerabilities, using Trivy [3]

In chapter 6, where artifact attestations are analyzed, we will demonstrate how their are leveraged in GitHub Actions as well.

[2] https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/scanners/sonarscanner-for-maven/

[3] https://github.com/aquasecurity/trivy-action

4 Sigstore

In this chapter we will analyze Sigstore. It is used by GitHub Actions to sign the artifact attestations we declare in our CI/CD pipelines and provides significant security benefits.

4.1 What is Sigstore

Problem statement: how can I sign artifacts, but drop the need for managing cryptographic material while at the same time tying the artifact to an OIDC identity, in an auditable manner?

Let's discuss this issue. In reality, any developer can write some code and upload it to a package repository (e.g., PyPI, npm, maven central). Then this package can be used as a dependency in another project. But no one asks about the code's underlying functionality or the identity of the publisher. The problem lies exactly there. The software supply chain is vulnerable to attacks originating from compromised packages. A mere dependency which contains malicious or vulnerable code can have a major impact for multiple targets concurrently. Package signing, is a solution for this problem, nonetheless, it has seen minimum adoption by its audience.

There are a few solid reasons as to why this happens. First of all, one must manage the cryptographic keys needed to sign the package. Practically, this poses an adoption burden. Keys have to be generated (probably manually), then do the signing process, and lastly store the private key securely. Private key compromised? Need to spend time to rotate. Secondly, one ought to be a *"trusted"* entity in order to have their package downloaded. The Web of Trust model doesn't really define a reason as to why someone should trust a public key uploaded on the Internet. Trust is built upon trust, that is, I trust someone because that someone is trusted by other people I trust. Additionally, there has to be some built-in functionality in the package manager that promotes package signing, and make it easier for end users to sign their code. Lastly, when someone downloads a package which is signed, the signature must be verified otherwise there was no reason to sign the package in the first place. Package managers can also help in that, e.g., when executing an *npm install* command, the signature can automatically get verified.

Into some statistics [9], a 2016 analysis of PyPI indicated that 4% of the repos list a signature, while 0.07% of users downloaded the signature for verification. Another study on RubyGems revealed that only 1.6% of the latest version of packages are signed.

Sigstore is a general purpose artifact signing system, which aims in minimizing that adoption barrier, while boosting the security aspects of the signing flow [9]. As an open source project, it has gained attention by the community, including big tech players as well who contribute to its development and maintenance.

How does Sigstore manage to drop the need for managing keys? The solution is very intuitive actually, it does that by deleting the private key after signing. If there is no private key, then there is no need to fear about getting compromised. Sigstore maintains two separate, immutable, publicly accessible and auditable trancparency logs. One is named ***Rekor*** (the artifact log) and the other is ***Fulcio*** (the identity log) which in reality are merkle trees. If one wants to sign an artifact using Sigstore, first he needs to "log into Sigstore" via OIDC. Fulcio acts as a Certificate Authority (CA) as well, apart from being a ledger. Three IdPs are supported as of now

- Microsoft
- GitHub
- Google

That implies that one must posses credentials for one of these IdPs in order to use Sigstore. After getting authenticated, Fulcio registers a record about the identity authenticated. Then, the user generates a key-pair and sends the public key to Fulcio. Fulcio, as a CA, generates a short-lived certificate (10 minutes), sends it back to the client, and appends the log with the newly created certificate. The client signs the artifact, packs a special packet called *envelope*, sends it to Rekor, and deletes the private key. Finally, Rekor creates a record which contains the artifact signature, the public key, claims about the subject authenticated and some other metadata about the artifact.

Now, whoever wants to verify the artifact signature can lookup to Rekor[1], find the record (e.g. search by artifact hash), download the signature and the public key and verify. And that is a core point in Sigstore. Keys are **ephemeral**. After signing an artifact, we only need to verify. If we need to sign again, we re-run the flow with **new** keys. The authors of Sigstore also call this scheme ***keyless signing***. For the exact same reason, certificates are short-lived. We want to ensure that only in a specific **small time frame** a private key is eligible to sign an artifact.

4.2 System Design

The system design is best described by the following image:

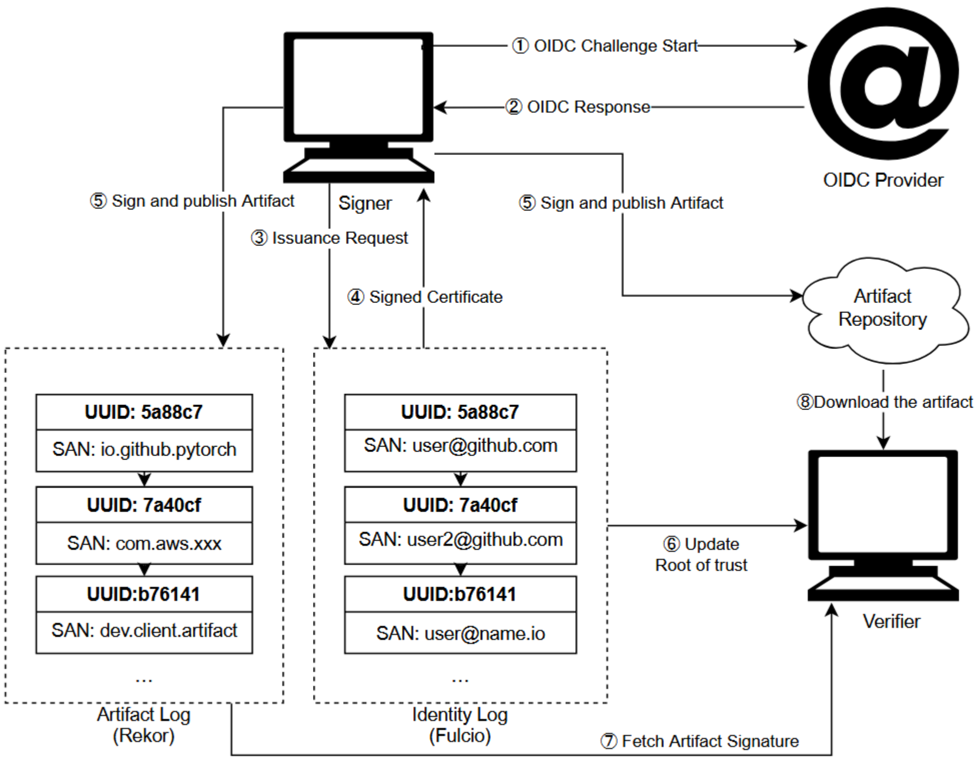


Figure 4.1: The Sigstore key issuance flow (source: Sigstore: Software Signing for Everybody [9])

[1]https://search.sigstore.dev/

A revisit of the steps in the picture:

1. In order to sign, Cosign[2] client asks for OIDC authentication.
2. OIDC provider authenticates cosign.
3. Cosign generates key-pair and sends a certificate signing request to Fulcio.
4. Fulcio replies with a 10-minute valid certificate and appends the certificate to its log.
5. Cosign signs the certificate, pushes the envelope to Rekor, and the artifact to an artifact repository (e.g., a container image to DockerHub). Artifact publishing could be a separate step in the image, since it is not done by cosign.
6. In order to verify, the client updates Fulcio's root of trust (we will not analyze how Fulcio acts as a CA[3]).
7. Get the record from Rekor which corresponds to the artifact in question.
8. Get the artifact from its repository, verify the signature and compare the hashes to check the authenticity and integrity.

4.2.a Cosign

Cosign is a simple tool to use:

```
cosign sign-blob ./sample.txt  # sign a local artifact
cosign sign myaccount/myimage:latest  # sign an image in a repo

cosign verify myaccount/myimage:latest
    --certificate-oidc-issuer-regexp=.*
    --certificate-identity-regexp=.*
```

Cosign is what communicates with the other Sigstore services, that is Fulcio and Rekor. In our use case, there are premade actions in the GitHub Actions marketplace which integrate with our CI/CD pipeline. For example:

[2]Along with Fulcio and Rekor, a client has been implemented specifically for sigstore which is called *cosign*. The "signer" in image 4.1 most likely runs the cosign client (we will refer to the signer as "cosign")

[3]More on that here

```yaml
name: Build Image
...
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Install Cosign
        uses: sigstore/cosign-installer@v3.7.0
        ...

      - name: Build and push Docker image
        uses: docker/build-push-action@v6.9.0
        with:
          tags: myaccount/myimage:latest
          file: ./Dockerfile
           push: true

      - name: Check install!
        run: cosign --yes sign myaccount/myimage:latest
```

4.2.b Fulcio and Rekor

As said before, both Fulcio and Rekor are transparency logs. Their scheme resembles a lot the one of Certificate Transparency (CT), where certificates are logged on a public ledger. Let's explore their properties:

1. Those two logs are immutable, and cryptographically verifiable since their data structure is a merkle tree.
2. They are both append-only
3. They are auditable, and actually there are parties within the community that lookup the logs for potential malicious activity
4. There is a Sigstore Public Good Instance managed by OpenSSF [2], that's why its infrastructure (the two ledgers) are open and accessible by everyone

Fulcio is the identity log and a CA. Being a CA is a necessity for issuing the short-lived certificates. Typically, when a certificate expires, it is decoupled from the subject that it was tied to. For instance, when a web server's public key certificate expires, the public key is no longer supposed to be trustworthy. However, with Sigstore the opposite is true. We maintain the certificate in the log as proof of the authentication event and for audits.

Rekor is the artifact log. No artifact is actually uploaded on this log, meaning, files, images or whatever else needs to be signed, does **not** exist in Rekor. The artifacts are uploaded in artifact repositories such as DockerHub, Artifact Hub etc. The records of Rekor **link** the artifact to a signature, its public key, and the subject that signed it. Why the subject? Because authentication happens via OIDC, as described in chapter 2, a set of claims is returned to the client (cosign). Fulcio first uses these claims to issue the certificate. Rekor comes second to log the subject's information. Now, even if a bad actor signs an artifact he is "**brought in public** auditable space" which speeds up forensic analysis if needed. At the same time, this poses a privacy issue for all users. Rekor logs the email of the cosign user. Since the ledger is immutable, what is written cannot be unwritten.

A scneario which showcases the previous statement, is a typosquatting attack. If I authenticate with the email *someone@gmail.com* then an adversary can try to login with the email *somone@gmail.com* and sign a different version of the same artifact I sign. Also, for each record in Rekor, there should be a corresponding one in Fulcio. Monitoring and running audits can capture this kind of activity.

It is important to note that Sigstore **trades the key management problem with the identity problem**. *"I can prove ownership because I am the holder of a private key"* vs*"I can prove ownership because I can authenticate via OIDC"*. The authors of Sigstore ran same interviews [9] which uncovered that the trade of key management problem for the identity problem is preferred. Package managers such as npm have also made an attempt to integrate Sigstore into their functionality.

5 Software Supply Chain Security

In this section we will study the concept of software supply chain, and describe the attack surface.

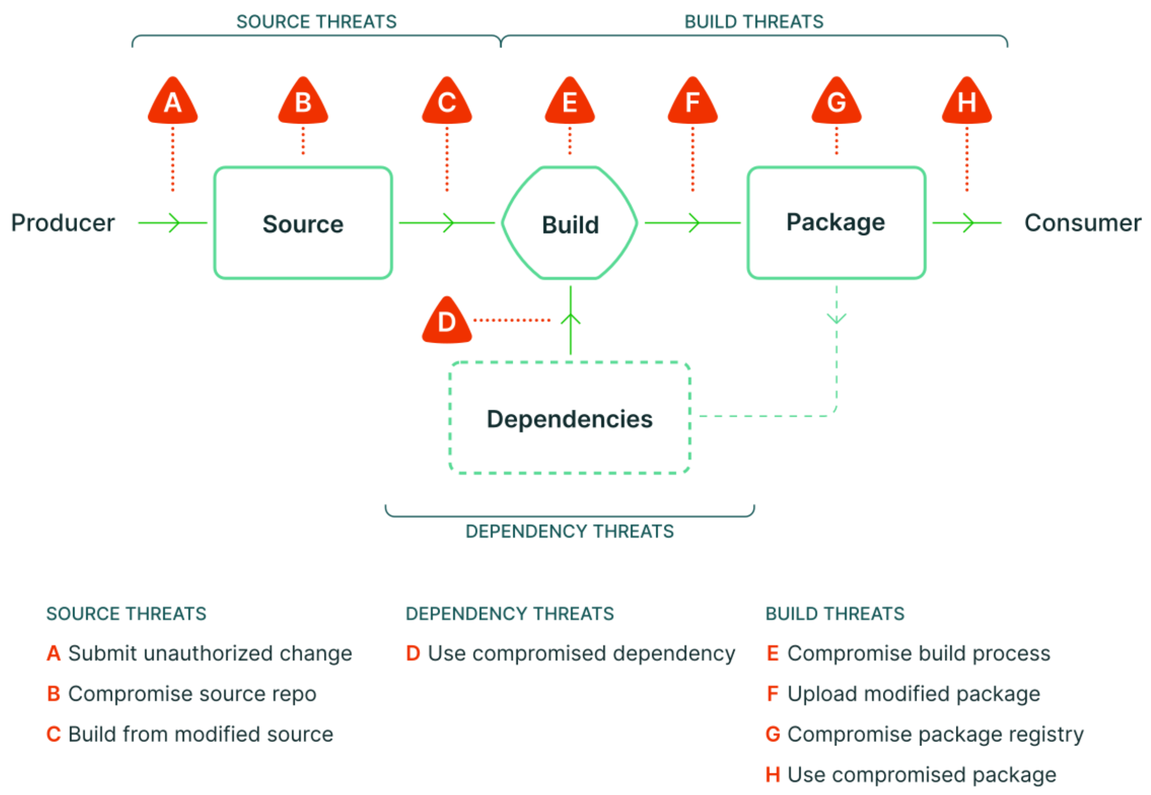

SOURCE THREATS
BUILD THREATS
A
B
C
E
F
G
H
Producer
Source
Build
Package
Consumer
D
Dependencies
DEPENDENCY THREATS
SOURCE THREATS
A Submit unauthorized change
B Compromise source repo
C Build from modified source
DEPENDENCY THREATS
D Use compromised dependency
BUILD THREATS
E Compromise build process
F Upload modified package
G Compromise package registry
H Use compromised package

Figure 5.1: Supply Chain Threats (source: SLSA [11])

Typically, the term *supply chain* refers to the logistics process of acquiring some materials from a producer, process it to create a product, and distribute it to its consumers. Similarly, in software supply chain a product owner will require software (the "materials"), produce custom software, and distribute it so that it reaches its clients. The problem is that this process can be disrupted by malicious actors. Image 5.1 demonstrates the supply chain attack surface:

The following table contains an example case for each of the image's threats:

Submit unauthorized change (to source repo)	SushiSwap: Contractor with repository access pushed a malicious commit redirecting cryptocurrency to themself.
Compromise source repo	PHP: Attacker compromised PHP's self-hosted git server and injected two malicious commits.
Build from modified source (not matching source repo)	Webmin: Attacker modified the build infrastructure to use source files not matching source control.
Use compromised dependency (i.e. A-H, recursively)	event-stream: Attacker added an innocuous dependency and then later updated the dependency to add malicious behavior. The update did not match the code submitted to GitHub (i.e. attack F).
Compromise build process	SolarWinds: Attacker compromised the build platform and installed an implant that injected malicious behavior during each build.
Upload modified package (not matching build process)	CodeCov: Attacker used leaked credentials to upload a malicious artifact to a GCS bucket, from which users download directly.
Compromise package registry	Attacks on Package Mirrors: Researcher ran mirrors for several popular package registries, which could have been used to serve malicious packages.
Use compromised package	Browserify typosquatting: Attacker uploaded a malicious package with a similar name as the original.

Table 5.1: Real world attacks (source: SLSA [11])

Supply chain security is indeed a difficult security aspect to address since it takes into consideration the whole SDLC. Developers embed dependencies into their code to help them create the functionality they need, and most often those dependencies are open source and maintained by people whose trustworthiness is - or should be - questionable. Even though the Free and Open-Source Software (FOSS) ecosystem promotes code reviews, they require a lot of manual effort which poses an economic burden for many organizations. Code written by developers itself may also not follow security best practices and thus, introduce application-level vulnerabilities. Then comes the build system. As discussed in chapter 3, code goes through a CI/CD pipeline in order to reach the end goal of delivering and deploying software. What if, during the build step, a malicious dependency tries to interact with the build system and read sensitive information (e.g., credentials)? Or, what happens if the compiler is compromised?

In their work Enck & Williams [4] have pointed out 5 specific challenges in supply chain security:

- updating vulnerable dependencies
- leveraging SBOMs
- choosing trusted supply chain dependencies
- securing the build process
- getting industry-wide participation

Say, the current version of a software contains a **vulnerable dependency**. The developers patch it and release a new version of it which isn't vulnerable. Is there any guarantee that the new version is *safer* than the previous one? Is there enough quality and functional testing done to ensure that the vulnerability is eliminated, or did the developers introduce a new bug/vulnerability in the software? Ideally, you will not want to be the first to update to a newer version, but give it some time to verify that the dependency can be trusted again.

Some organizations are skeptical about the usage of **SBOMs**. On the one hand, some believe that it is useless since vulnerabilities are context-specific. Sharing the information that a vulnerable package version exists in a software does not imply that it exploitable as well. On the other hand, it is argued that the widespread adoption of SBOMs provides useful insights, confidence, transparency, and paves the way to zero-trust security.

There is no real definition to *being trusted*, as **trust cannot be quantified**. Any entity can claim to be trusted. Likewise, someone can vouch for a dependency's trustworthiness, but trust might as well be shared with the people that back up this dependency. A repo can change maintainers through its life cycle, and some of them may be unreliable. Best practices state that one could fork the official repo of a dependency and embed it in their version control system. That way, there is more control over the code of the dependency, it can be reviewed, cleaned, or modified. This is called *vendoring* [1].

For **securing the build process** itself, there is a standard called SLSA which defines 4 levels of security for the build process. Lower levels can contain no or just a few security measures such as generating provenance attestation, while higher levels are more complicated using reproducible[1], hermetic builds etc. Reproducible builds provide bit-to-bit accuracy when trying to replay a build, but considering the inherent non-determinism of the build process, it is a hard goal to achieve.

Lastly, there is a need to raise awareness in the community. Supply chain security will be on the edge of research for the next 10 years as said by Enck & Williams [4]. There have been initiatives by the academic and industry sectors to provide the community with tools and guidance on how to face supply chain security. Examples of that is Sigstore, in-toto and SLSA.

So, the key takeaways for supply chain security are:

1. how do I bring in dependencies
2. how do I secure the build process
3. who do I consider to be trusted

[1]Effort has been initiated to dive deeper into reproducible builds, see here for more

6 Artifact Attestations

In this chapter we will dive deep into artifact attestations by first looking at two important specifications (in-toto and SLSA), and lastly provide an example use case.

6.1 in-toto

in-toto is a framework which sets the baselines for supply chain security. It describes how software should be developed, tested, built and packaged [20]. It assures the client that the software in question has gone through integrity and authenticity checks by looking at the supply chain. Through in-toto, the supply chain processes have become transparent and can be audited by downstream consumers of an artifact. The main goals of in-toto are:

1. verify that the software has gone through all the appropriate steps of the supply chain
2. verify the software's owner

Practically, in-toto enforces a software owner to create a ***layout*** (i.e., define an order of steps in the chain), define the input (***materials***) and output (***products***) for each step, and who is **authorized** to perform this step. That *"who"* is the ***functionary*** in in-toto terminology. Each step produces a metadata file (***link file***) which is the exact same file used by the client for verification. in-toto does **not** prevent unauthorized or malicious action but rather cares about if all steps in the chain where executed by the one(s) who must execute it, with the expected input and output results. In other words, it protects the supply chain from actions **other than the ones intended to run**.

The three main components of in-toto are:

- a tool used by the software owner to create supply chain layouts
- a tool that functionaries can use to create link metadata about a step
- a tool the clients can use to verify the final product

We will gain better understanding of those components by looking at the file formats of in-toto. All of them are primarily JSON files[1]. Let's begin with the layout:

```
{
  "_type" : "layout",
  "expires" : "<EXPIRES>",
  "readme": "<README>",
  "keys" : {
     "<KEYID>" : "<PUBKEY_OBJECT>"
  },
  "steps" : [
    "<STEP>",
    "..."
  ],
  "inspect" : [
    "<INSPECTION>",
    "..."
  ]
}
```

The layout is signed by a trusted key which belongs to the software owner. *_type* is the type of the file, in this case it's "layout". *Expires* is the expiration date of the layout. *Readme* is an optional field which contains human readable text about the supply chain. *Keys* is a list of all the public keys used in the steps section. *Steps* is a list of steps and restrictions that apply to them. It can also be a *sub-layout* which contains multiple steps. *Inspect* is a list of suggested actions and commands to run on the client side for verification.

The following is the steps file format:

```
{
    "_type": "step",
```

[1]All of the following JSON files and example are taken from the in-toto specification [20]

```
        "name": "<NAME>",
        "threshold": "<THRESHOLD>",
        "expected_materials": [
           [ "<ARTIFACT_RULE>" ],
           "..."
        ],
        "expected_products": [
           [ "<ARTIFACT_RULE>" ],
           "..."
        ],
        "pubkeys": [
           "<KEYID>",
           "..."
        ],
      "expected_command": "<COMMAND>"
  }
```

In this case *_type* is set to "steps". *Name* is the name of the step e.g., build, test, etc. *Threshold* is an integer which indicates how many link metadata file must be used to verify this specific step (links will be showcased later in this section). *Expected materials* defines what input must be passed into this step, for it to be considered legitimate. In in-toto actually, there is a specific syntax for writing rules about the aforementioned, but this is out of scope for this thesis. *Expected products* similarly is a list of rules which define what the output of the step should be. *Pubkeys* is a list of key ids which belong to the specific functionaries who are authorized to run this step. Lastly, *expected command* is a list of commands that was run to execute this step. Note that this should **not** be trusted for verification purposes.

Then there is the inspection file:

```
  {
        "_type": "inspection",
        "name": "<NAME>",
        "expected_materials": [
           [ "<ARTIFACT_RULE>" ],
           "..."
        ],
        "expected_products": [
           [ "<ARTIFACT_RULE>" ],
        "..."
     ],
     "run": "<COMMAND>"
 }
```

Here, the consumer is in possession of the software product and reads the inspection file to see what actions must be done to verify it. *Name* is a name for the inspection. *Expected materials* and *expected products* are similar to the steps file. *Run* contains the command to run.

Lastly, there is the link file:

```
{
    "_type" : "link",
    "name" :  "<NAME>",
    "command" : "<COMMAND>",
    "materials": {
       "<ARTIFACT_NAME>": "<HASH>",
       "..." : "..."
    },
    "products": {
       "<ARTIFACT_NAME>": "<HASH>",
       "..." : "..."
    },
    "byproducts": {
      "stderr": "",
      "stdout": "",
      "return-value": null
    },
    "environment": {
      "variables": "<ENV>",
      "filesystem": "<FS>",
      "workdir": "<CWD>"
    }
}
```

A link file is produced in each step ran by the corresponding functionaries. *Name* must be the same as with the step it relates to. *Command* is the command ran by the functionary in this step, along with its arguments. *Materials* and *products* are same as before, and are referenced by their hash. *Byproducts* are not verified in reality by in-toto, but can provide useful insights while inspecting the step (e.g., what was printed in the terminal when this step was executed?). *Environment* contains information about the environment in which the step was ran e.g., environment variables, working directory etc.

For tooling there is a cli tool[2] which can be used to sign, verify metadata, and other actions as well.

6.1.a Example Workflow

We will now see how in-toto is practically used by providing a use case and some exemplary JSON files. The example is taken as-is from in-toto specification [20].

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.

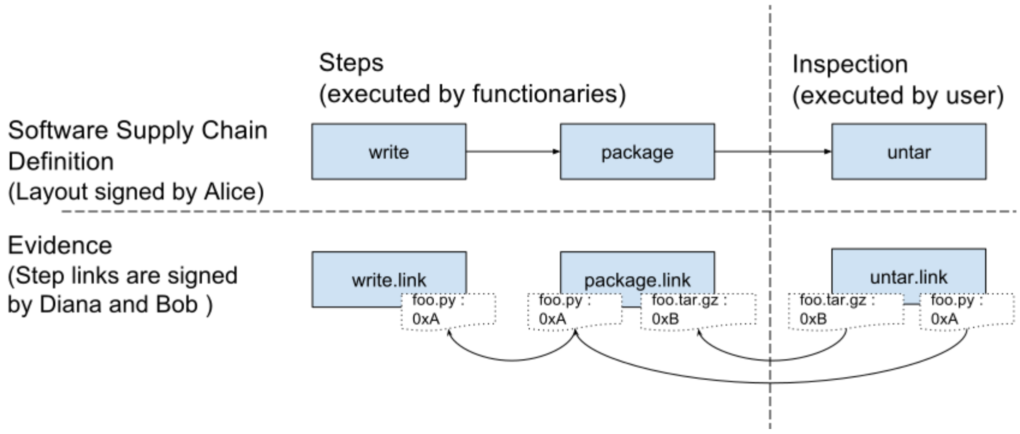


Figure 6.1: Supply chain pieces for this example (source: in-toto specification [20])

[2]Example use for cli tool can be found here and its documentation here

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>",
  "readme" : "<README>",
  "keys" : {
      "<BOBS_KEYID>" : "<BOBS_PUBKEY>",
      "<ALICES_KEYID>": "<ALICES_PUBKEY>"
   },
  "steps" : [
    { "_type": "step",
      "name": "write-code",
      "threshold": 1,
      "expected_materials": [],
      "expected_products": [
        ["CREATE", "foo.py"]
      ],
      "pubkeys": [
        "<ALICES_KEYID>"
      ],
      "expected_command": ["vi"]
    },
    { "_type": "step",
      "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": [
    { "_type": "inspection",
      "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>",
    "sig" : "90d2a06c7a6c2a6a93a9f5771eb2e5ce0c93dd580be..."
  }
]

```

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 (let write-code.link and package.link):

```
{ "signed" : {
  "_type" : "link",
  "name": "write-code",
  "command" : ["vi", "foo.py"],
  "materials": { },
  "products": {
    "foo.py": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d..." }
  },
  "byproducts": {
    "stderr": "",
    "stdout": "",
    "return-value": 0
  },
  "environment": {
    "variables": [""],
    "filesystem" : "",
    "workdir": ""
  }
},
"signatures" : [
  { "keyid" : "<ALICES_KEYID>",
    "sig" : "94df84890d7ace3ae3736a698e082e12c300dfe5aee92e..."
    }
  ]
}
```

```
{ "signed" : {
    "_type" : "link",
    "Name": "package",
    "command" : ["tar", "zcvf", "foo.tar.gz", "foo.py"],
    "materials": {
      "foo.py": { "sha256": "2a0ffef5e9709e6164c629e8b31bae0d..."}
    },
    "products": {
      "foo.tar.gz": { "sha256": "78a73f2e55ef15930b137e43b9e..."}
    },
    "byproducts": {
      "stderr": "",
      "stdout": "foo.py",
      "return-value": 0
    },
    "environment": {
      "variables": [""],
      "filesystem" : "",
      "workdir": ""
    }
  },
  "signatures" : [
    { "keyid" : "<BOBS_KEYID>",
      "sig" : "ae3aee92ea33a8f461f736a698e082e12c300dfe5022a..."
      }
    ]
}
```

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 artifact 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.

6.1.b in-toto attestation framework

Up until now we have explained why in-toto adds transparency to the supply chain, and provided a use case as well. Now we will introduce the attestation framework of in-toto which is what matters to us. Note that the attestation framework is a separate specification. "The in-toto Attestation Framework provides a specification for generating verifiable claims about **any aspect of how a piece of software is produced**" as said by the authors [19]. Why is this different from the core in-toto specification analyzed above? Now, with the attestation framework the focus is on verifying artifacts produced amid the supply chain, **not** on verifying the chain itself.

The attestation framework [19]

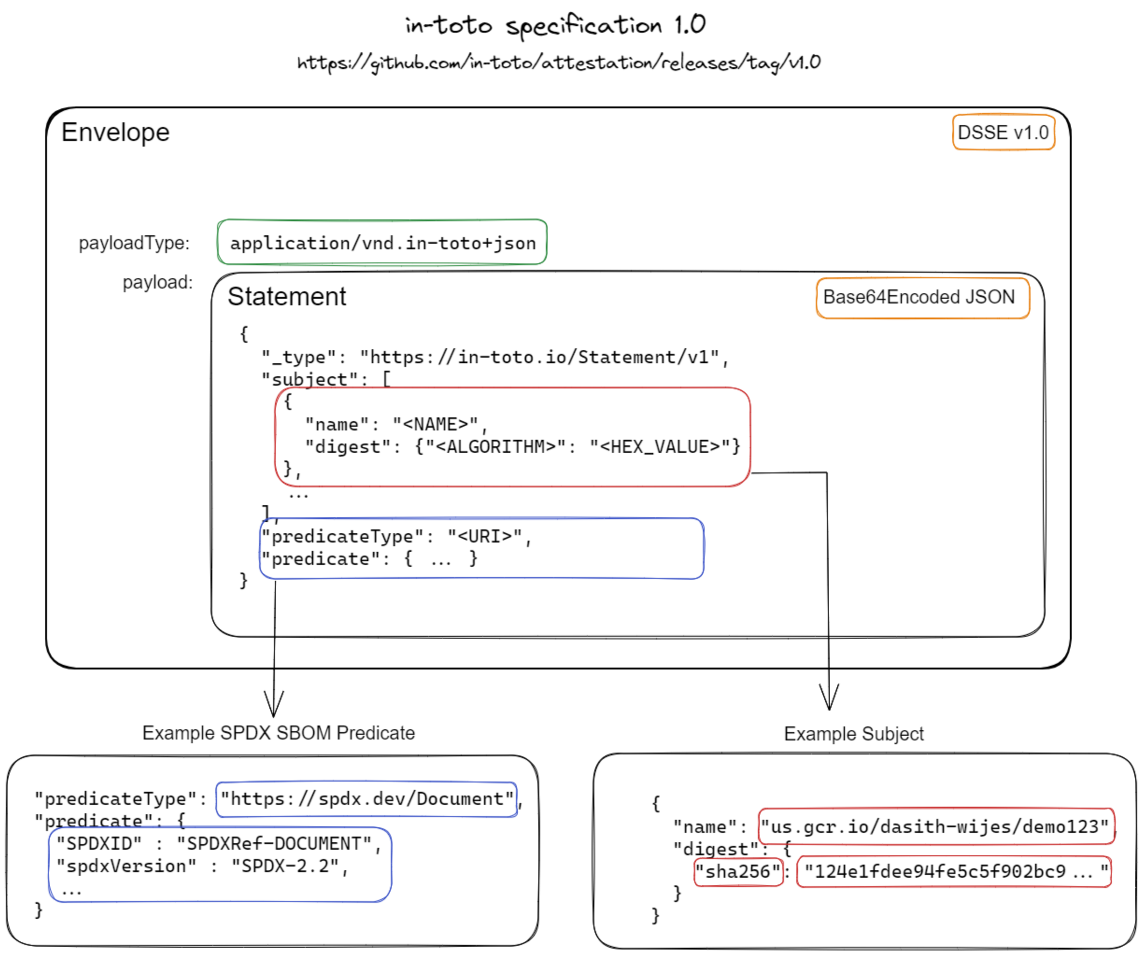
- defines a standard format for attestations which bind subjects, the artifacts being described, to arbitrary authenticated metadata about the artifact
- provides a set of pre-defined *predicates* for communicating authenticated metadata throughout and across software supply chains

This framework is intended to sign arbitrary artifact metadata so that any policy engine can consume it. By standardizing the way in-toto attestation are produced and consumed, we can leverage them in CI/CD pipelines for both purposes i.e., create an attestation while running the pipeline, and verify before consuming. To achieve this, the community has vetted for specific attestations which are thought to be commonly used. Here is a list of some of them:

- SLSA Provenance
- Link
- SCAI Report
- Runtime Traces
- SLSA Verification Summary

We can take for example the vulnerability scans. Frequently when creating a container image, you may want to run a vulnerability scan on it e.g., with Trivy (see chapter 3.3). There may be cases where you may want to **attest** that there are no vulnerabilities of high or critical severity residing in this container. For that a vulnerability scan attestation can be created. Now, let's take a look at how this attestation is formatted.

Attestations are primarily JSON files. In in-toto attestation framework, an attestation is packaged in the following way:


in-toto specification 1.0
https://github.com/in-toto/attestation/releases/tag/v1.0
Envelope
DSSE v1.0
payloadType:
application/vnd.in-toto+json
payload:
Statement
Base64Encoded JSON
Example SPDX SBOM Predicate
Example Subject

```
{
  "_type": "https://in-toto.io/Statement/v1",
  "subject": [
    {
      "name": "<NAME>",
      "digest": {"<ALGORITHM>": "<HEX_VALUE>"}
    },
    ...
  ],
  "predicateType": "<URI>",
  "predicate": { ... }
}
```

```
"predicateType": "https://spdx.dev/Document",
"predicate": {
  "SPDXID" : "SPDXRef-DOCUMENT",
  "spdxVersion" : "SPDX-2.2",
  ...
}
```

```
{
  "name": "us.gcr.io/dasith-wijes/demo123",
  "digest": {
   "sha256": "124e1fdee94fe5c5f902bc9 ... "
  }
}
```

Figure 6.2: Relationships between the envelope, statement and predicate layers (source: in-toto attestation specification [19])

Attestations are packed as a *DSSE* (explained in 2.5) with payloadType being "application/vnd.in-toto+json". If we decode from base64 the payload, then we get a *Statement*. A Statement has the following format:

```
{
  "_type": "https://in-toto.io/Statement/v1",
  "subject": [
    {
      "name": "<NAME>",
      "digest": {"<ALGORITHM>": "<HEX_VALUE>"}
    },
    ...
  ],
  "predicateType": "<URI>",
  "predicate": { ... }
}
```

For a Statement, the *_type* is "https://in-toto.io/Statement/v1". The *subject* is the artifact that it refers to, and is referenced by its digest. Then there is the *predicateType* and *predicate*. The *predicate* is actually the body of the attestation, and contains whatever information this attestations specifies, for instance, a vulnerability scan attestation may have a filed named "results" and list the vulnerabilities found. *PredicateType* is a URI which indicates the schema of the *predicate*. In the aforementioned example, it would be "https://in-toto.io/attestation/vulns", so that the application knows how to interpret the predicate, and what fields to expect.

There are multiple predefined predicate types in the specification which correspond to common attestations. If a predicate type does not fit your needs, you can always implement your own. Here are some notable ones [19]:

- CycloneDX or SPDX → for SBOM attestation
- SLSA Provenance → for provenance attestation
- SCAI Report → for evidence-b ased assertions about software artifact and supply chain attributes or behavior
- Vulns → for results of vulnerability scans on artifacts
- Runtime Traces → for capturing runtime traces of software supply chain operations
- Test Result → for expressing results of any type of tests (e.g. mvn test)

In the image below we see an example of a CI/CD pipeline, which depicts the supply chain of a project.

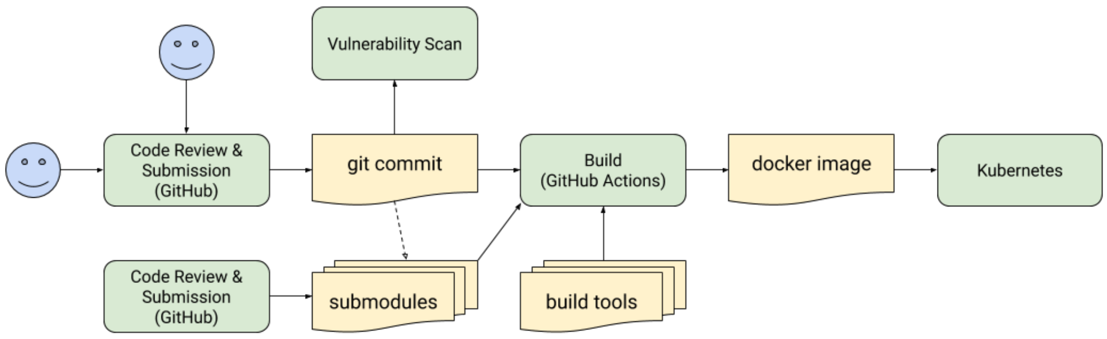


Figure 6.3: Attestation supply chain (source: in-toto attestation documentation [18])

Of course, one can use the in-toto cli tool to verify that all the steps in the supply chain were executed by the authorized functionaries, but now with in-toto attestation framework in each step an attestation can be generated for a specific artifact, and have it verified upon consuming it. We can have a code review attestation, a provenance attestation, a vulnerability scan attestation and a release attestation for this example. For simplicity we list one Statement of the ones mentioned:

```
{
  "predicateType": "https://example.com/VulnerabilityScan/v1",
  "subject": [
    { "digest": {
        "gitCommit": "859b387b985ea0f414e4e8099c9f874acb217b94"}
    }
  ],
  "predicate": {
    "timestamp": "2020-04-12T13:55:02Z",
    "vulnerability_counts": {
      "high": 0,
      "medium": 1,
      "low": 17
    }
  }
}
```

When the above Statement is packed into a DSSE, it is an attestation. That is because attestations must contain a signature for integrity and authentication purposes. Now imagine, that we have more of these Statements, with different predicate types such as, "https://example.com/**CodeReview**/v1" or "https://example.com/**Provenance**/v1". This patent helps strengthen the security of the supply chain, and provide confidence to downstream consumers of the software artifacts produced by it. At this point we need to highlight that attestations should be stored in a (public) repository in order for them to be discoverable and usable.

6.2 Supply Chain Levels for Software Artifacts (SLSA)

Supply Chain Levels for Software Artifacts or SLSA, is a set of "adoptable guidelines for supply chain security, established by industry consensus" [11], SLSA is a specification which is separate from in-toto but they share some intersection points (analyzed in subsection 6.2.d). SLSA is about protecting each step in the supply chain by providing tamper-resistant proof (i.e., attestations). SLSA helps protect against [11]:

- Code modification (by adding a tamper-evident “seal” to code after source control)
- Uploaded artifacts that were not built by the expected CI/CD platform (by marking artifacts with a factory “stamp” that shows which build platform created it)
- Threats against the build platform (by providing “manufacturing facility” best practices for build platform services)

While SLSA as a specification claims to protect the whole chain, practically in the current version 1.0, it mainly focuses in the build process of the software product. It works very tightly with CI/CD platforms which automate the build process and make it easier to efficiently produce attestations.

SLSA can be used by three distinct parties. Software producers, software consumers and infrastructure providers. Software producers want to generate the tamper-proof evidence that their chain is secured by the three threats mentioned above. Software consumers want to verify this proof to evaluate their trust against a software product. Infrastructure providers want to embed some baseline security measures into their infrastructure - like CI/CD platforms - which securely host the pipelines of the software producers. For infrastructure providers this is especially important since they act as the bridge between software producers and consumers, and their services must be as trustworthy as possible. This correlates to the concept of *workload identities* as mentioned in 3.2.b.

6.2.a Provenance

Provenance is a core concept of SLSA. The provenance of an artifact depicts what system built it, what process was used and what where the inputs [11]. For example, a provenance for a maven build would have a value for the property of "system", say, "github runner" and for "inputs", say, "app.jar, commit a923f.." etc. GitHub Actions has a premade action for generating provenance attestations. Below we can see an example of it:

```yaml
name: Build with provenance

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          java-version: '17'

      - name: Maven Package
        run: mvn clean package

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3.7.1

      - name: Build and push Docker image
        uses: docker/build-push-action@v6.9.0
        with:
          tags: myaccount/myimage:latest
          file: ./Dockerfile
          push: true

      - name: Attest Build Provenance
        uses: actions/attest-build-provenance@v1.4.3
        with:
          subject-name: docker.io/myaccount/myimage:latest
          subject-digest: ${{steps.build-and-push.outputs.digest}}
          push-to-registry: true
```

The above workflow is similar to the ones in chapter 3, but in the last step we have added a provenance attestation. What this exactly does is, it collects information about the build environment and the artifact in scope and crafts a JSON file. This JSON file is in reality an in-toto attestation with a *predicateType* of "https://slsa.dev/provenance/v1". Moreover, to sign this attestation the specific GitHub Action leverages Sigstore as a signing system [14]. The identity that signs the provenance attestation is the GitHub runner which is a workload identity, and uses Sigstore with non-interactive OIDC authentication to sign the document. Sigstore's ephemeral keys are a great fit for this case because the workload identity ceases to exist after the pipeline is executed. What is the **information gain** from a provenance attestation? That a particular build platform produced a set of artifacts [11]. Below we can see a sample provenance attestation produced by this GitHub action:

```
{
  "_type": "https://in-toto.io/Statement/v1",
  "subject": [
    {
      "name": "docker.io/myaccount/myimage:latest",
      "digest": {
          "sha256": "2366a3df5da25590e07ecbc..."
      }
    }
  ],
  "predicateType": "https://slsa.dev/provenance/v1",
  "predicate": {
    "buildDefinition": {
      "buildType": "https://actions.github.io/buildtypes
                                                /workflow/v1",
      "externalParameters": {
        "workflow": {
          "ref": "refs/heads/main",
          "repository": "https://github.com/myaccount/myproject",
          "path": ".github/workflows/ci.yaml"
        }
      },
      "internalParameters": {
        "github": {
          "event_name": "push",
          "repository_id": "123456789",
          "repository_owner_id": "32165487",
          "runner_environment": "github-hosted"
        }
      },
      "resolvedDependencies": [
        {
          "uri": "git+https://github.com/myaccount
                      /myproject@refs/heads/main",
          "digest": {
                "gitCommit": "3456cva5b3gfv9s8..."
          }
        }
      ]
    },
    "runDetails": {
      "builder": {
        "id": "https://github.com/myaccount/myproject/.github
                          /workflows/ci.yaml@refs/heads/main"
      },
      "metadata": {
        "invocationId": "https://github.com/myaccount
              /myproject/actions/runs/12345678912/attempts/1"
      }
    }
  }
}
```

6.2.b How SLSA works

Now that we have a clear understanding of what provenance is for SLSA, let's see how SLSA itself works. SLSA works in tracks and levels. A track is an aspect of the supply chain e.g., build process, and for each track there are incrementing levels of security that it provides. The higher the level, the more security practices are implemented for this track. Currently, SLSA supports one track which is the Build track and has four levels of security as seen in table 6.1. The build track's focus is oriented around the ***provenance*** of an artifact.

Track/Level	Requirements	Focus
Build L0	(none)	(n/a)
Build L1	Provenance showing how the package was built	Mistakes, documentation
Build L2	Signed provenance, generated by a hosted build platform	Tampering after the build
Build L3	Hardened build platform	Tampering during the build

Table 6.1: SLSA Build Levels and Tracks (source: SLSA [11])

Build Level 0 provides no guarantees and is intended for software builds under the development phase where no evidence needs to be provided to a downstream consumer.

Build Level 1 includes provenance of an artifact and can be used to avoid trivial mistakes but is easy to tamper with. It is intended for organizations who want to quickly get into SLSA and gain some benefits from provenance. These benefits are:

- makes it easier to debug or analyze the build process by knowing exactly what the build environment was
- with verification errors can be avoided such as building from a commit that is not present in the upstream repo
- an inventory of software and build platforms is assembled

Level 1 provenance may not be signed or lack some fields in its document. It is required by the software producer to have a consistent build process - i.e., have CI pipeline in place - so that stakeholders will know what to expect from a typical build.

Build Level 2 states that hosted build platform must be used e.g., GitHub Actions hosted runners, that generates and signs the provenance. The build platform should run on dedicated infrastructure, not an individual's workstation, and the provenance is tied to that infrastructure through the digital signature. Since security levels are incremental, the Level 2 includes all Level 1 benefits plus:

- prevents tampering **after** the build $\rightarrow$ logical since provenance is signed
- reduces attack surface by limiting builds to specific build platforms that can be audited and hardened
- allows large-scale migration of teams to supported build platforms early while
- deters adversaries who face legal or financial risk by evading security controls, such as employees who face risk of getting fired

Build Level 3 is the upper level of security and is all about hardening the build platform. Forging a provenance or evading verification requires exploiting a vulnerability that is beyond the capabilities of most adversaries. That means that the build platform itself must be resilient do prevent tampering **during the build**. The build platform should implement strong controls to prevent runs from influencing one another and prevent secret material used to sign the provenance from being accessible to other user-defined build steps. To correlate with the example yaml given in 6.2.a, the last point means that the key generated by the step "Attest Build Provenance" must not be accessible by any other following step (assuming there are following steps such as vulnerability scans etc). Benefits gained from Level 3 are the ones accumulated from the levels below, plus:

- prevents tampering **during** the build by insider threats, compromised credentials, or other tenants.
- greatly reduces the impact of compromised package upload credentials by requiring attacker to perform a difficult exploit of the build process.
- provides strong confidence that the package was built from the official source and build process

So what is left now, is to have build platform providers harden their systems, while at the same time encourage software producers to adopt a well-implemented build platform, and apply SLSA provenance attestations.

6.2.c SLSA Principles

The fact that a build is required to run on a (hardened) platform implies that a software producer ***trusts*** the infrastructure provider that the builder is honest and won't act maliciously. That is logical because at some point one has to place trust somewhere in order to get the job done. To generalize, SLSA has highlighted three principles [11]:

- Trust platforms, verify artifacts
- Trust code, not individuals
- Prefer attestations over inferences

The first principle is safe to say that is quite reasonable and was explained beforehand. Only a **small** number of build platforms should be trusted by software producers. That is because the trust is put on the infrastructure provider. It is preferable to trust an organization that is well-known for its security practices - like GitHub - and use their services, rather than use any build platform with questionable security implementations for their systems. On verification, it is easy to check if the artifact was produced by one of those "trusted" platforms. If the infrastructure does not belong to a third party, but it is on-prem, then it is better to have all parties interested in running build on one shared dedicated environment and harden it, rather than having each party maintain their own ones.

The second principle is also logical. Source code is static and analyzable so problems can be easily traced back to source. Individuals on the other hand can be adversaries, or even if they aren't, they are prone to mistakes. Even if that is not the case, they tend to ask for access here and there and end up being authorized in services they shouldn't be.

Lastly, it is said to prefer attestations over inferences. One can make assumptions about the security of a system by looking at its configurations. Is this switch turned off? Who is allowed access here? Does the build step generate provenance? And so on. Even an experienced person can skim a list with these properties and indeed conclude that the system is safe, but this is just not acceptable. We need better means of inference, and SLSA promotes attestations for that. With that, only authorized entities can make changes to artifacts that are being attested.

6.2.d SLSA & in-toto

SLSA and in-toto may be very similar at first glance but serve different purposes. Nonetheless, there are intersection points.

SLSA is a specification that states what levels of security can be achieved for specific tracks of the supply chain, for example, what security measurements can be taken to protect and prevent from malicious action during and/or after the build process. It is a set of guidelines for infrastructure providers, devops engineers, security engineers and other roles, to have in mind when trying to implement security solutions in their supply chain.

in-toto is a specification for making the supply chain transparent, and making sure that no unauthorized functionaries attempt to run a step they shouldn't. Additionally, attestations have been introduced in in-toto which help verify the authenticity and integrity of artifacts produced in the chain. The framework has also standardized the format for these attestations. There are multiple attestations types such as CycloneDX, Vulns and SLSA Provenance.

What SLSA gets from in-toto is the standardized way of producing attestations in a specific format. On the other side, what in-toto gets from SLSA, is the predicate and predicateType of SLSA provenance. As said, in in-toto the community vets for predicate types to be added in the specification, and SLSA provenance is one of them, along with SLSA verification summary attestation.

6.3 Verification of attestations

SLSA sets some guidelines about verifying artifacts which are all about checking specific fields on provenance attestation. The steps to do that are the following [11]:

1. Check SLSA Build level
2. Check expectations
3. (Optional) Check dependencies recursively

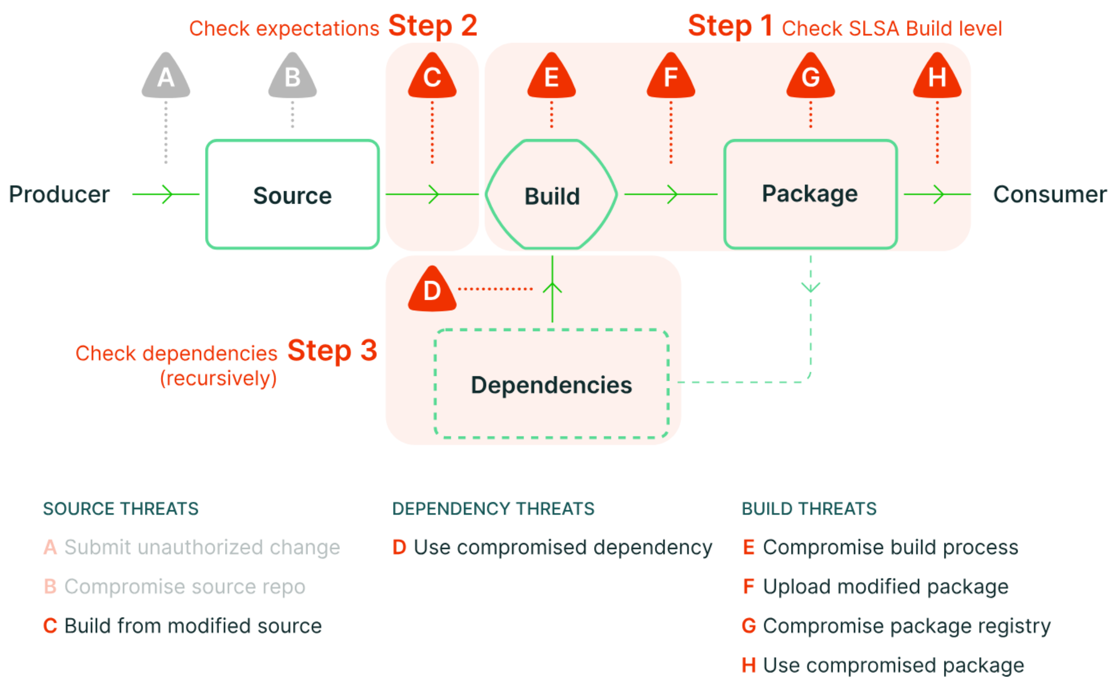

Check expectations Step 2
Step 1 Check SLSA Build level
A
B
C
E
F
G
H
Producer
Source
Build
Package
Consumer
D
Check dependencies Step 3 (recursively)
Dependencies
SOURCE THREATS
A Submit unauthorized change
B Compromise source repo
C Build from modified source
DEPENDENCY THREATS
D Use compromised dependency
BUILD THREATS
E Compromise build process
F Upload modified package
G Compromise package registry
H Use compromised package

Figure 6.4: Protection of each step in the chain after verification (source: SLSA [11])

We briefly explain the steps. In **step 1** we check that the build level actually applies to the artifact by checking the provenance. At the same time, the builder's trust is evaluated (not all build platforms are/should be trusted). The predicate type must be "https://slsa.dev/provenance/v1", the subject must match the artifact being verified and the attestation must be successfully verified. Not all levels defend against all the steps in the chain in image[3] 6.4.

In **step 2** the verifier can set some expectations about the provenance of an artifact. That is because, in step 1 not all fields of the attestation are verified. In this step, a consumer could set an expectation for the source repository, that is, to see whether the repository in the attestation matches the expected one (i.e., the real one) and there has not been a typo-squatting attack. At this point, the verifier is supposed to have a predefined set of meaningful expected values to verify, according to a specific threat model. Within the context of a predefined policy, a verification can fail or succeed.

In **step 3** the *resolvedDependencies*[4] field of the provenance is recursively checked.

[3]For more detail, see here

[4]resolvedDependencies is a field in provenance which contains a collection of artifacts needed at build time

Until now we have seen the *SLSA way* of verifying attestations which tells us what to check during verification. GitHub proposes its own cli tool to verify [14]. When an attestation is created with GitHub Actions, it is stored in a temporary storage location (a URL for that is provided when the action is executed). When having possession of the artifact we can run the following command to verify:

```
$ gh attestation verify my-artifact.tar.gz -o my-organization
```

An alternative proposed is to download the attestation document, not the artifact itself, and pass the attestation to a policy engine like Open Policy Agent (OPA), and let the policy engine inference the authenticity of the artifact. A policy engine like OPA contains policies which are written by individuals, and have expected values hard-coded in them. This can be done using the cli ad-hoc as well:

```
$ gh attestation verify -R github/example --format json myfile.zip | \
opa eval  --stdin-input -f raw \
-d policy.rego "data.attestation.slsa1.allow == true"
```

7 Use Case

7.1 Description

In our use case, we have a Java Quarkus application which is implements a REST API for car rental service[1]. This is not a monolithic application, but it is split in three microservices running in a Kubernetes cluster:

- Fleet microservice
- User microservice
- Rent microservice

Each microservice is responsible for different functionalities of the application. *Fleet microservice* manages the vehicles that each company rents e.g., a new vehicle can be added for a company. *User microservice* does user management e.g., registration. *Rent microservice* executes mostly application logic e.g., by accepting requests for car rentals by customers, accepting payments etc. Each microservice has its own H2 database. In the image below we can see the topology of the cluster:

[1]Project can be found here. It started as a monolithic app, but later on changed to microservices

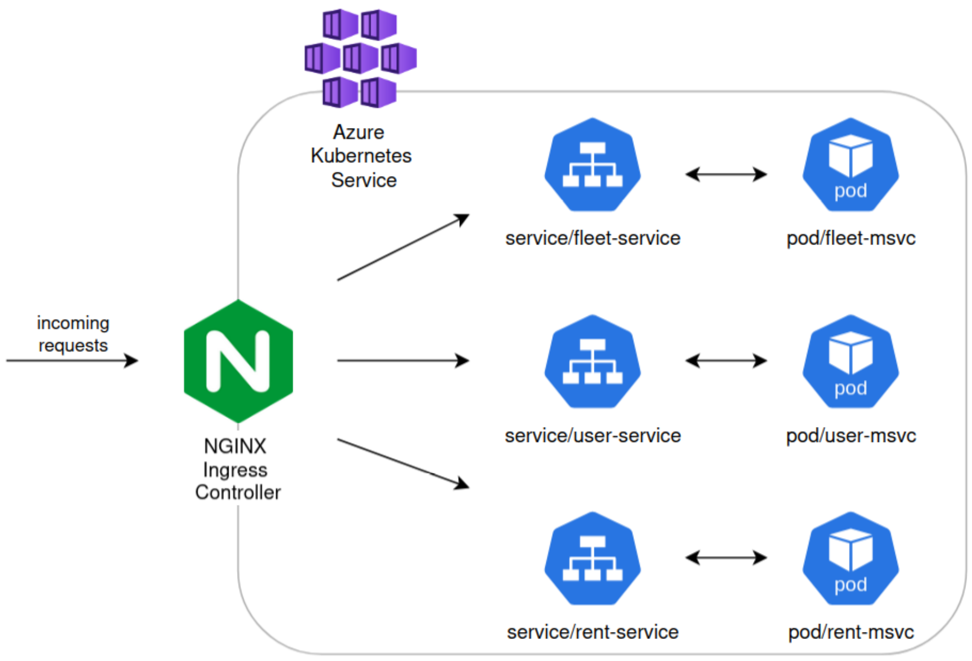


Figure 7.1: Kubernetes cluster topology

NGINX is used as an Ingress Controller for the cluster. It accepts requests and forwards the to the corresponding service. Note that this application was not built with security in mind, so we do not care about application-level or API-level security aspects. Terraform is used to provision the infrastructure needed, while within the Kubernetes cluster there are multiple other services running like Istio, OPA, and telemetry services such as Jaeger and Prometheus. Those services are out of scope since we want to explore artifact attestations, but are mentioned for completeness.

7.2 DevSecOps applied

This project has adopted DevSecOps principles and uses GitHub Actions as a build platform to test, build, and distribute its containers, and later on deploy the containers. GitHub container registry is used for storing the containers, and GitHub artifact registry for storing the attestations. After the CI pipeline is executed, the CD pipeline takes place which is a push-based deployment to AKS. A high level overview of the pipeline is depicted in the following image:

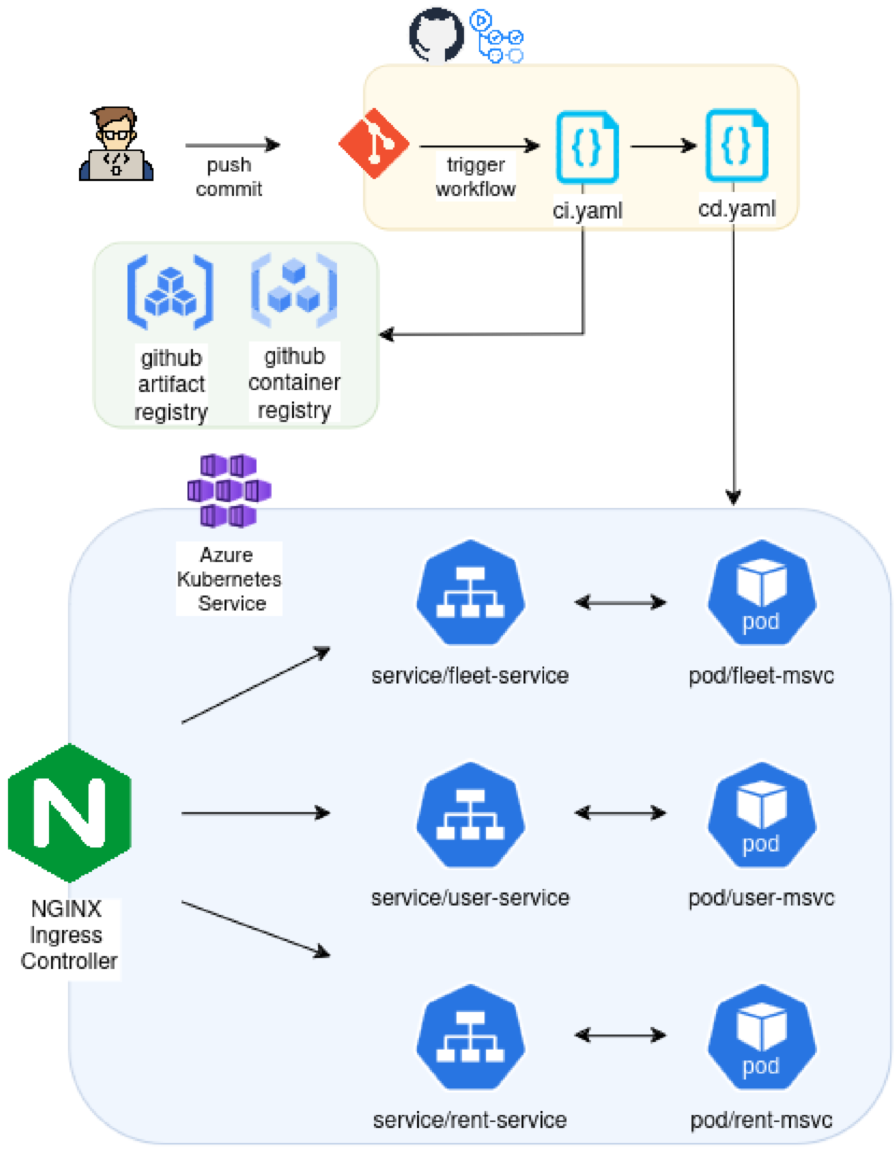


Figure 7.2: Project pipeline

The actions the **CI** process incorporates are:

- checkout and install prerequisites (java, maven etc)
- testing
- code quality scanning with Sonarqube
- building
- packaging
- creating the container

- **attest its provenance**
- generate and **attest the SBOM**
- vulnerability scan with Trivy (no attestation is done here)

Let's assume that the pipeline runs for the *fleet* microservice. We would have a yaml file looking like this one (for simplicity we skip steps that have been mentioned multiple times until now):

```yaml
name: Build Fleet Microservice

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      contents: write
      packages: write
      id-token: write
      attestations: write

    steps:
      # checkout, install java & maven
      ...
      - name: Build and analyze
        env:
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN_FLEET }}
          SONAR_HOST_URL: ${{ secrets.SONAR_HOST_URL }}
        run: mvn -B --file ./FleetMicroservice/pom.xml verify
       org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
       -Dmaven.test.skip=true
       -Dsonar.projectKey=quarkus-rest-car-rental-service-fleet
       -Dsonar.projectName='quarkus-rest-car-rental-service-fleet'

      # build & push container image
      ...
      - name: Attest Build Provenance
```

```
        uses: actions/attest-build-provenance@v1.4.3
        with:
          subject-name: ghcr.io/lefosg
                    /quarkus-rent-car-rental-service/fleet:latest
          subject-digest: ${{steps.build-and-push.outputs.digest}}
          push-to-registry: true

      - name: Generate SBOM
        uses: anchore/sbom-action@v0
        with:
          image: ghcr.io/lefosg
                    /quarkus-rent-car-rental-service/fleet:latest
          output-file: fleet.sbom.json
          format: 'cyclonedx-json'
          upload-artifact: false

      - name: Attest SBOM
        uses: actions/attest-sbom@v1
        with:
          subject-name: ghcr.io/lefosg
                    /quarkus-rent-car-rental-service/fleet
          subject-digest: ${{steps.build-and-push.outputs.digest}}
          sbom-path: fleet.sbom.json
          push-to-registry: true

      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@0.28.0
        with:
          image-ref: ghcr.io/lefosg
                    /quarkus-rent-car-rentl-service/fleet:latest
          format: 'table'
          ignore-unfixed: true
          vuln-type: 'os,library'
          output: fleet.trivy
```

The first security step that appears in the pipeline is the SAST scanning with Sonarqube. This step takes as input our code, analyses it, and sends a report to a specified Sonarqube server defined in the secret variable *secrets.SONAR_HOST_URL*. There, we can see a full report of the code quality, for example if there are any known vulnerabilities in our code e.g., SQL injection.

Next, there is the action *actions/attest-build-provenance* which is the one that generates the build provenance. The container is already uploaded in the GitHub container registry. As explained in 6.2.a, the signing happens via Sigstore, and once the process is done, the artifact attestation is uploaded to GitHub's artifact registry. After the workflow is done running a URL is provided to access the document. Same happens with SBOM attestation. First we have to generate the SBOM by running the *actions/attest-sbom* action and the create its corresponding attestation. The action *actions/attest-sbom* takes as input a JSON file which contains the SBOM, and generates its attestation by signing it. Afterwards, the SBOM is uploaded to GitHub's artifact registry and similarly to the provenance attestation, it is accessible by a URL provided.

Lastly, the action *aquasecurity/trivy-action* runs the Trivy vulnerability scanner on the microservice container. It looks for vulnerabilities in the container image, misconfigurations, files and secrets. It outputs a file in the format specified, in our case *table*.

Note, that in the *permissions* of the job in the yaml we have set *packages: write* and *attestations: write*. We specifically need these two permissions to be able to upload the container image to the GitHub packages service, and upload the artifact attestation to the artifact registry.

Now, we will have a look at the **CD** pipeline. The steps are fairly simple:

- checkout the repository
- verify the provenance attestation of the container about to be deployed
- log in to Azure
- configure the Kubernetes context
- apply the yaml that includes the corresponding container image (push-based deployment)

Again, let's assume that we are talking about the Fleet microservice pipeline. Here is a demo yaml file:

```yaml
name: Deploy Fleet Microservice

on:
  workflow_run:
    workflows: [ "Build Fleet Microservice" ]
    types:
      - completed

jobs:

  deploy-fleet:
    permissions:
      actions: read
      contents: read
      id-token: write
    runs-on: ubuntu-latest

    steps:

      - uses: actions/checkout@v4

      - name: Log into registry ghcr.io
        uses: docker/login-action@v3.3.0
        with:
          registry: 'ghcr.io'
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Verify Rent Container Attestation
        run: gh attestation verify oci://lefosg
            /quarkus-rest-car-rental-service/fleet --owner github

      # Logs in with your Azure credentials
      - name: Azure login
        uses: azure/login@v1.4.6
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}

      # Use kubelogin to configure kubeconfig
      - name: Set up kubelogin for non-interactive login
        uses: azure/use-kubelogin@v1
        with:
          kubelogin-version: 'v0.0.25'

      # Retrieves AKS cluster's kubeconfig file
      - name: Get K8s context
        uses: azure/aks-set-context@v3
        with:
          resource-group: 'car-rental-serice-rg'
          cluster-name: 'car-rental-service-cluster'
          admin: 'false'
          use-kubelogin: 'true'

      # Deploys Fleet Microservice
      - name: Deploys application
        uses: Azure/k8s-deploy@v4
        with:
          action: deploy
          manifests: ${{ github.workspace }}/k8s/fleet-msvc.yaml
          images: ghcr.io/lefosg/quarkus-rest-car-rental-service
                        /fleet:latest
```

The steps are pretty much self explanatory. Login to GitHub container registry and Azure, configure the *kubectl* client to point to AKS, and apply the yaml file in the last step. The one step that is of most interest is the one that runs the *gh attestation verify* command. This command runs some checks predefined by the *gh* cli tool, like checking that the repository is indeed the correct one.

In the Kubernetes cloud-native environment attestations can be verified in a "Kubernetes-like way". We can always run the *gh attestation verify* command to manually verify the container image attestation, but in reality, we can install a specific Kubernetes Admission Controller which knows how to verify attestations, based on a custom-written policy that the Kubernetes administrator composes and applies. GitHub has its own Admission Controller [23]. It combines two Helm charts which are installed in our Kubernetes cluster. All we have to do is write a yaml file which is the *ClusterImagePolicy* file provided by the trust-policy helm chart. This yaml is our policy definition, and whenever a change is applied to the cluster, say, a new container is entering the cluster, the Kubernetes Admission Controller pulls the attestation from GitHub and runs the checks specified in our custom ClusterImagePolicy yaml.

8 Conclusion and Discussion

To conclude, in this thesis we discussed various topics. From the fundamentals of DevSecOps, up until artifact attestations and their supporting frameworks. DevSecOps is an important concept, and it is crucial for organizations who do DevOps engineering to understand the risks of the supply chain and embed security practices in their workflows. As for artifact attestations, the specifications of in-toto and SLSA are indeed a good headway to lay the foundations of supply chain security but they still need time and effort to grow to mature specifications. The same goes for existing technical tools. in-toto has implemented a cli tool for its purposes however there is a need for more streamlined solutions. Other tools like policy engines and CI/CD platforms need to be "supply-chain aware" and integrate with upcoming solutions.

Bibliography

[1] URL: https://cloud.google.com/software-supply-chain-security/docs/dependencies.

[2] Evan Anderson. *Running Sigstore as a Managed Service: A Tour of Sigstore's Public Good Instance*. Oct. 2024. URL: https://openssf.org/blog/2023/10/03/running-sigstore-as-a-managed-service-a-tour-of-sigstores-public-good-instance/.

[3] Krebs Bruno. *The OpenID Connect Handbook*. Tech. rep. Auth0. URL: https://assets.ctfassets.net/2ntc334xpx65/2yRtkzYHiiBLLSguFsnQs9/419405cee8bd0a7b8f70e20cef22c190/The-openid-connect-handbook-v1.pdf.

[4] William Enck and Laurie Williams. "Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations". In: *IEEE Security Privacy* 20.2 (2022), pp. 96–100. DOI: 10.1109/MSEC.2022.3142338.

[5] Artem Golubev. *What Is CI/CD? Expedite The Software Development Life Cycle - testRigor*. June 2022. URL: https://testrigor.com/blog/what-is-cicd/.

[6] *Information technology — Object Management Group Architecture-Driven Modernization (ADM) — Knowledge Discovery Meta-Model (KDM)*. Standard. Geneva, CH: International Organization for Standardization, 2012. URL: https://www.iso.org/standard/32625.html.

[7] Ramtin Jabbari et al. "What is DevOps? A Systematic Mapping Study on Definitions and Practices". In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. Edinburgh, Scotland, UK: Associa-

tion for Computing Machinery, 2016. ISBN: 9781450341349. DOI: 10.1145/2962695.2962707. URL: https://doi.org/10.1145/2962695.2962707.

[8] Julia Kulla-Mader and Chuck Lantz. *Apply software engineering systems*. July 2024. URL: https://learn.microsoft.com/en-us/platform-engineering/engineering-systems.

[9] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. "Sigstore: Software Signing for Everybody". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 2353–2367. ISBN: 9781450394505. DOI: 10.1145/3548606.3560596. URL: https://doi.org/10.1145/3548606.3560596.

[10] Marc Ohm et al. "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Clémentine Maurice et al. Cham: Springer International Publishing, 2020, pp. 23–43. ISBN: 978-3-030-52683-2. DOI: 10.1007/978-3-030-52683-2_2.

[11] OpenSSF. *Supply-chain Levels for Software Artifacts*. 2022. URL: https://slsa.dev/.

[12] Roshan N. Rajapakse et al. "Challenges and solutions when adopting DevSecOps: A systematic review". In: *Information and Software Technology* 141 (2022), p. 106700. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106700. URL: https://www.sciencedirect.com/science/article/pii/S0950584921001543.

[13] RedHat. *What is a CI/CD pipeline?* May 2022. URL: https://www.redhat.com/en/topics/devops/what-cicd-pipeline.

[14] Trevor Rosen. "Blocked Page". In: *github.blog* (2024). URL: https://github.blog/news-insights/product-news/introducing-artifact-attestations-now-in-public-beta/.

[15] Trevor Rosen. "Where does your software (really) come from?" In: *The GitHub Blog* (Apr. 2024). URL: https://github.blog/security/supply-chain-security/where-does-your-software-really-come-from/.

[16] secure-systems-lab. *dsse/envelope.md at v1.0.0 · secure-systems-lab/dsse*. 2020. URL: https://github.com/secure-systems-lab/dsse/blob/v1.0.0/envelope.md.

[17] *Storing and sharing data from a workflow*. 2024. URL: https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/storing-and-sharing-data-from-a-workflow.

[18] in-toto. *attestation/docs/README.md at main · in-toto/attestation*. 2021. URL: https://github.com/in-toto/attestation/blob/main/docs/README.md.

[19] in-toto. *attestation/spec/v1.0/README.md at v1.0 · in-toto/attestation*. 2021. URL: https://github.com/in-toto/attestation/blob/v1.0/spec/v1.0/README.md.

[20] in-toto. *specification/in-toto-spec.md at v1.0 · in-toto/specification*. 2024. URL: https://github.com/in-toto/specification/blob/v1.0/in-toto-spec.md.

[21] Dimitri Van Landuyt et al. *From automation to CI/CD: a comparative evaluation of threat modeling tools*. 2024-07-23.

[22] Ryan Wike et al. *Workload identities - Microsoft Entra Workload ID*. Oct. 2023. URL: https://learn.microsoft.com/en-us/entra/workload-id/workload-identities-overview.

[23] April Yoho. *Configure GitHub Artifact Attestations for secure cloud-native delivery*. July 2024. URL: https://github.blog/security/supply-chain-security/configure-github-artifact-attestations-for-secure-cloud-native-delivery/.

Acronyms

AKS	Azure Kubernetes Service. 15, 56
CA	Certificate Authority. 22
CD	Continuous Delivery. 13, 56, 60
CI	Continuous Integration. 12, 56, 57
CI/CD	Continuous Integration/Continuous Delivery. v, 1, 2, 29, 63
CT	Certificate Transparency. 25
DAST	Dynamic Application Security Testing. 17
DevOps	Development and Operations. v, 1, 11, 12, 63
DevSecOps	Development, Security, and Operations. v, 17, 56, 63
DSSE	Dead Simple Signing Envelope. 9, 41
FOSS	Free and Open-Source Software. 29
IdP	Identity Provider. 6
JWS	JSON Web Signature. 9
JWT	JSON Web Token. 7
OIDC	OpenID Connect. 6, 21, 22

OPA Open Policy Agent. 53, 56

RP Relying Party. 6

SAST Static Application Security Testing. 17

SCA Software Composition Analysis. 17

SDLC Software Development Life Cycle. 1, 5, 12, 28

SLSA Supply-chain Levels for Software Artifacts. v, 30, 63

TM Theat Modeling. 17

VM Virtual Machine. 14

List of Figures

List of Tables

List of Algorithms