RESEARCH PROJECT 1

Analysis of a rarely implemented security feature: signing container images with a Notary server

January, 2021

Students:
Mohanad Elamin
melamin@os3.nl

Rio Kierkels
rkierkels@os3.nl

Abstract

Ensuring the integrity of software is a fundamental and essential function of software update systems. However, lacking or sometimes even missing implementations of this by organizations has lead to a number of significant hacks in the past. Some industry efforts are ongoing in this area, though this paper focuses on a single project called Notary. Notary is a tool to publish and manage trusted content collections by implementing The Update Framework, a framework for securing software update systems. Container image distribution, while different than more traditional operating system package distribution, has the same fundamental integrity needs. Docker, who originally built Notary before donating the project to the Cloud Native Computing Foundation, has integrated image signing and verification functionality in its command line tool making use of the Notary project. We aim to uncover some of the recommended practices are when using Notary as a container image signing solution. By building a set of Kubernetes manifests that attempt to be a good starting point for production deployments of Notary, we will build up our knowledge of the system. A number of compromise scenarios, like key compromise, have been executed using this setup. With this exercise we have uncovered some challenges with deploying and operating the system which we think contributes to it's relatively low adoption in the industry. On top of that the focus of it's main developers has switched to a version 2 of the system. This is noticable by the lack of a recent release, the latest of which at the time of writing was in April of 2018. Even though Notary works as advertised and hits their main goal of securing container images using signatures, the industry and the people involved seem to be holding their breath while the design of Notary Version 2 is being fleshed out. For anyone with more pressing needs for an image signing solution, this paper and the associated manifests will give more insight into Notary's concepts and operational properties helping you make a more informed decision.

1 Introduction

Content trust is one of the fundamental security concerns for any container-based system. Before deploying any container image, the system must establish the integrity and authenticity of its content. Without validation and verification, an adversary might exploit the inherently trusted relationship between the vendor and the user and may cause severe damage to organizations. A recent example of a software supply chain cyberattack is the compromise of the update process of SolarWinds Orion IT system management platform [1]. By inserting malicious and unauthorized code into one of the software update packages, the adversaries managed to exploit the system to allow remote access into the environment.

The significance of securing the software update process led to numerous industry and academic space efforts to solve that challenging dilemma. A very well received and adopted project created to tackle software updates' security challenges is The Update Framework [2]. The Update Framework is an open-source project founded by Justin Cappos of the Secure Systems Lab at New York University. The technology behind The Update Framework is a result of work done by Justin Cappos and Justin Samuel at the University of Washington as well as Nick Mathewson and Roger Dingledine of The Tor Project [3]. Since its inception, The Update Framework, also known as TUF, is adopted by multiple technology companies and organizations in various implementations, including Docker Notary [4]. However, in 2017, the Cloud Native Computing Foundation adopted both the Docker Notary project and The Update Framework [5]

Docker Notary is an implementation of The Update Framework based on the Go language. The Notary project version one outlined multiple vital goals, including *Survivable Key Compromise* to protect against various key compromise scenarios and *Freshness Guarantees* to protect against replay attacks. Another objective of the Notary project is providing *Configurable Trust Thresholds* to protect against malicious content publishing in the case of the loss of individual or group of signing keys. Moreover, Notary also provides *Signing Delegation* to add flexibility, especially for large organizations. The *Use of Existing Distribution* is also an essential part of Notary. It allows the use of existing publishing channels. And finally, *Untrusted Mirrors and Transport* grant the feasibility of mirroring and distributing the Notary metadata via arbitrary channels. [6]

Apart from the primary use case of container distribution, Docker Notary has other production use cases. For example, Cloudflare uses Notary as part of the PAL container identity bootstrapping tool [7]. Furthermore, companies like Kolide use Notary for securing their automated software updates [8]. However, based on AWS's container security survey, the adoption rate for container image signing is still slow [9]. The AWS report shows that less than 10% of the participants are using container image singing using Notary. This research project will evaluate the likely reason behind the slow adoption, and it will also explore the challenges behind that and provide good practices and recommendations for production usage.

This paper's content includes the following chapters: Introduction, Research Questions, Background about The Update Framework and Docker Notary, Related Work, Methodology, Discussion, Conclusion, Future Work, Acknowledgment, and finally appendices.

2 Research Questions

The research project main question is defined as:

- **What are the best practices of using Notary for container image signing?**

To be able to answer the main research question, the following sub-questions are specified:

- How does Notary ensure the integrity and security of container images?
- What are the challenges of deploying Notary for container image signing?
- Based on the Proof of Concept test results, what are the main probable reasons of low adoption for Notary and container image signing?

3 Background

This section will provide a brief overview of Docker Notary and its underlying security framework, The Update Framework also known as TUF.

3.1 The Update Framework

The primary goal of The Update Framework is securing the software update system. In the paper, Survivable Key Compromise in Software Update Systems, Justin Samuel and Justin Cappos et al. (2010) [3] outlined four different design principles to be used as a framework for software update systems to provide resilience against signing keys compromises:

1. **Responsibility Separation**: The fundamental idea of responsibility separation is using different roles for distinct responsibilities, and each of the roles defines the exact actions allowed by a trusted party. The main advantage of role separation is to limit the attack surface in the case of a single role key compromise. The Update Framework adopts this principle and defines four primary top-level roles, each with its key and a metadata file [10]:

    - **Root role**: The root role serves as the root of trust, and it delegates trust to other top-level roles. Therefore, any organization employing TUF based systems must keep the root role's private key in a secure offline environment. The root role private key signs a root metadata file (root.json), which specifies the list of keys authorized for all four top-level roles. Revocation or replacement of any top-level roles keys is done by updating and signing the root metadata file using the root's private key.
    - **Targets role**: For the update system to securely present the trusted files to the end client, the metadata files describing the trusted files have to be signed using the targets role key. The trusted file metadata includes filenames, sizes, and corresponding hashes of those files. A target role can delegate trust, fully or partially, to another role to obtain further responsibility separation. Subsequently, a delegation role can pass the trust to other delegation roles.
    - **Snapshot role**: The snapshot role key signs a metadata file containing the latest version of the top-level target and delegation metadata. The snapshot role's primary goal is validating other roles' metadata integrity and preventing mix-and-match attacks. In the original paper of TUF this role was named the release role. However, to avoid ambiguity with the term release, the release role is renamed to the snapshot role. [11]
    - **Timestamp role**: A timestamp role key ensures freshness by regularly signing the hash of the latest snapshot metadata files along with a timestamp.

Figure 1 illustrates The Update Framework roles' hierarchy and the relationship between the metadata files, as explained in the TUF specification.

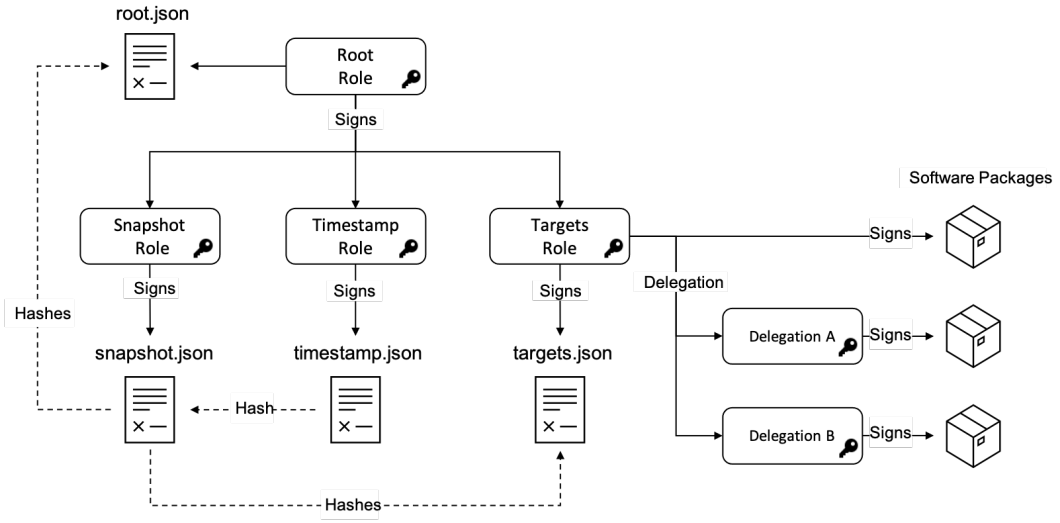

Figure 1: The hierarchy of The Update Framework roles

2. **Multi-signature Trust**: The use of multi-signature trust is recommended as an extra layer of defense against key compromise. Two approaches are available to achieve multi-signature trust, either by requiring a minimum t signers out of n potential signers from the same role, protecting against the compromise of *t-1* singer keys, or sharing responsibility across roles by requiring a signer from multiple roles. The two approaches are not mutually exclusive and can be combined.

3. **Explicit and Implicit Revocation**: Since signing keys in principle can be compromised, the update system must include methods for key revocation. The Update Framework divides revocation into two types; Explicit revocation triggered by notifying the clients to stop trusting keys or Implicit revocation by using expiration timestamp information as part of the roles corresponding metadata files. After the expiration or a specific number of usages, the key is considered untrusted.

4. **Minimizing Risk**: When discussing securing the software update systems, the primary risk factor is unauthorized usage of the signing keys. However, it is difficult to measure the impact of a single or multiple role compromise accurately. Accordingly, The Update Framework recommends minimizing the risks of key compromise by storing critical keys such as the root key in an offline system.

The Update Framework specification [10] outlines multiple implementation goals, including easy to implement client, a simple repository push process, and security within environments that don't support SSL. Another goal for The Update Framework is providing software update protection against a specific set of attacks. However, the framework attack protection doesn't guarantee update availability during an attack. Furthermore, one of The Update Framework's primary targets is eliminating the need for external Public Key Infrastructure and allowing full or limited trust delegation between the framework roles. On the other hand, The Update Framework doesn't define software packages formats and will not provide attack remediation or bootstrap security. Those capabilities are the responsibility of the software update system.

3.2 Docker Notary

Docker describes Notary as *"a tool for publishing and managing trusted collections of content."* [4]. Utilizing Docker Notary, software publishers can sign an arbitrary collection of data using The Update Framework as an underlying security model. Therefore, consumers can trust the signed content by validating the integrity and origin. Notary uses Globally Unique Names (or GUN for short) to uniquely identify collections of trust. For the context of signing container images, the GUN will include the registry, repository name, and the image tag.

Notary service architecture consists of two primary sub-systems, and both use a backend database for persistent storage:

- A **Notary server** that keeps and updates the TUF metadata of the trusted collection and ensure its validity. Notary server can optionally use JSON Web Tokens [12] to authenticate clients. Clients typically interact with the Notary server to request or upload metadata.
- A **Notary signer** is a system that stores the private signing keys and performs the actual signing operations following the Notary server requests.

For scalability or high availability, an organization can deploy multiple instances of both the Notary server and signer along with its corresponding database.

When using Docker Notary, the workflow of signing or validating content starts from a client. Multiple clients can interact with Notary; docker CLI, docker daemon, or Notary CLI. When using Docker Notary, the package signing process goes through the following steps [13]:

1. The client generates a metadata file for the package and uses the target key to sign it. Afterward, the client uploads the metadata file to the Notary server, optionally after authentication.
2. After receiving the metadata file, the Notary server validates the content checksum and signature. The server also checks its database for conflicting versions.
3. If sanity and validity checks succeed, the Notary server will generate timestamp and snapshot metadata files. The server will send the files to the signer for signing using the timestamp and snapshot keys.
4. The Notary signer will use the timestamp and snapshot keys to sign the metadata files and send the signature back to the Notary server.
5. The Notary server stores the timestamp and snapshot signatures in its database and sends a successful response to the client. The server uses the signed information to certify the target metadata as trusted and most recent for the collection.
6. Following the successful response, the client will immediately request updated target metadata from the server, including the timestamp. In the event of timestamp expiry, the Notary server will go through the same process again to generate and sign a new timestamp.

Figure 2 explains the communication steps between a client and the Notary Server and Signer. The diagram is an adapted version from Docker documentation [13].

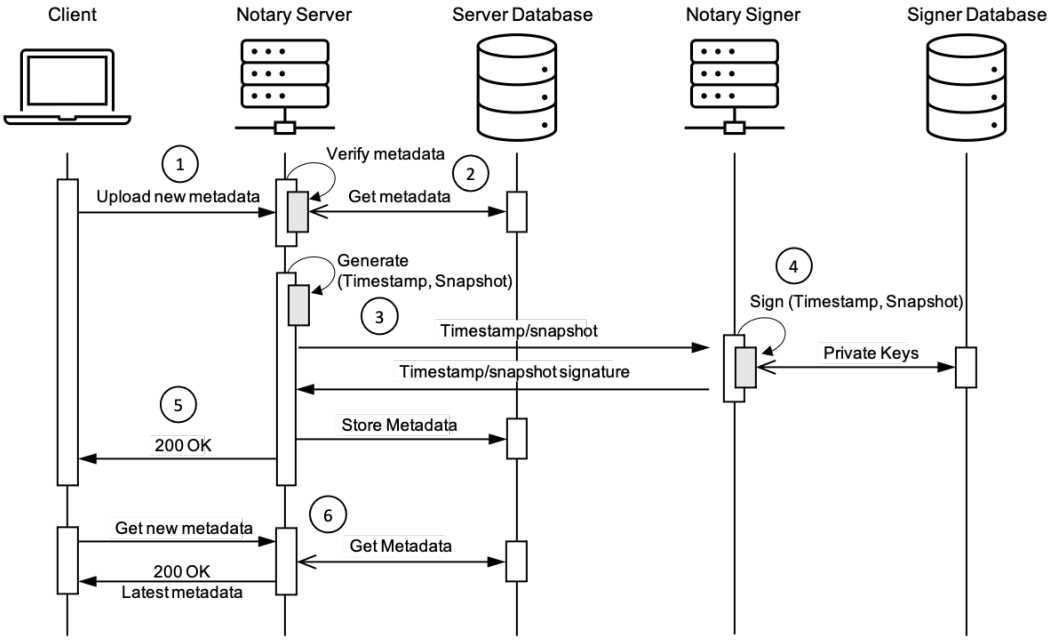

Figure 2: Notary Server and Signer communication

4 Related Work

Related analysis performed on the Docker Notary Project was mainly around five different write-ups. The first study is a paper from the SANS Institute, Authored by Stefan Winkel [14]. In that paper, Stefen discussed the Security Assurance of Docker Containers from the DevOps model's angle. Along with discussing the Docker Notary project, the report also investigated the security scanning of Docker images and the three aspects of Docker Security; Hardening, Patching, and Monitoring. Finally, Stefan also discussed container image security testing in different CI/CD pipeline stages. However, the discussions of Docker image scanning and testing are outside the scope of the current paper.

Besides the extensive report from SANS Institute, three penetration tests and vulnerability assessments performed on the Docker Notary software and The Update Framework:

1. **Docker Notary Application Penetration Test by the NCC Group** [15]: In July 2015, The NCC Group performed a Docker Notary penetration test as requested by the Docker company. In the report, the NCC Group focus was mainly on three core items: Notary Client/Server, Notary Signer, and the use of The Update Framework within Notary. The test result found multiple vulnerabilities in the Notary software, including two with high severity. As part of the report, NCC Group provided short-term and long-term solutions for each of the vulnerabilities. Based on the feedback from NCC Group, Docker patched most of the vulnerabilities.

2. **Pentest-Report TUF/Notary by Cure53** [16]: Cure53, a German cybersecurity firm, did another security assessment for the TUF and Notary software in 2018 as requested by The Linux Foundation. Cure53 methodology included a source code audit as well as penetration testing. In contrast to the NCC group's earlier test, Cure53 managed to reveal four limited severity issues. Cure53 team linked the positive results to the Cloud Native Computing Foundation's choice of using the Go language and the underlying libraries and tools used by both TUF and Notary projects.

3. **The Update Framework (TUF) Security Assessment by the NCC Group**

[17]: As requested by Kolide, an infrastructure analytics company, the NCC Group in 2017 performed another security assessment; however, this assessment concentrates mainly on The Update Framework's security rather than the Notary project. Kolide uses TUF and Notary for their software updater client [8]; Therefore, the scope of the assessment only verifies the Kolide TUF client against The Update Framework. The result of the testing activity resulted in two main improvement areas with no direct impact on the client. The first issue allows an attacker to write backups in unauthorized locations. On the other hand, the second issue enables a user to access updates without confidentiality and authenticity verification. As part of the report, the NCC Group provided recommendations to mitigate both problems and provide high-level guidance to enhance the client's overall security.

Aside from the Docker Notary and The Update Framework's security assessment and pen tests, other security vendors also shed some light on the importance of container image signing. An example is a write-up from TrendMicro, a cybersecurity software company, about the Docker Content Trust [18]. In the blog post, Brandon Niemczyk of TrendMicro discussed the building blocks of the Docker Content Trust or DCT and provided comprehensive implementation steps to enable it using Docker Notary. As part of the write-up, Brandon also explained the need for trust validation during the entire CI/CD pipeline. However, that is not part of the Docker Notary project scope.

5 Methodology

To analyze and demonstrate the properties of the Notary project we will be deploying it on top of Kubernetes [19]. This will help with reproducibility and gives us a singular language to talk about the deployment primitives. The Notary project itself provides different Docker Compose [20] setups to experiment with and can be found in their GitHub repository [6]. However these do not use the officially released binaries and container images but, instead, build Notary from source.

We will use the `Day 0`, `Day 1` and `Day 2` terminology, generally used to describe the phases in the software lifecycle, as a structure to talk about the different aspects of our Notary setup.

- **Day 0 - Design**: Here we will describe the architecture used for Notary along with the dependencies that are required and why they are needed. Scope will be discussed as well with regards to the security and reliability of those dependencies.
- **Day 1 - Deployment**: This is where we will actually deploy our architecture, verify that everything is running and validate that Notary functions as expected by signing, pushing and pulling an image. The manifests used will be provided in our GitHub repository [21] along with some scripts to help bootstrap a Kubernetes cluster locally (if desired), download some tools and apply the manifests.
- **Day 2 - Operations**: Finally we'll run a number of scenarios that could be encountered like key compromise and run through the steps required to mitigate these events.

5.1 Day 0 - Design

Our deployment and management substrate will be Kubernetes [19]. It's declarative and extensible API will help us with delegating certain tasks to other processes that usually would be executed by an administrator, one example would be certificate management. In theory any Kubernetes distribution could be used from production grade clusters (think cloud provider products) to locally hosted clusters running in Docker or directly on your

machine. For our experiment we'll be using k3d [22] which is a containerized version of k3s [23], a minimal distribution of Kubernetes meant to run on low powered hardware while still providing the full capabilities that Kubernetes has to offer. The only requirement for running it is that Docker has to be installed. A short list of locally runnable alternatives can be found in appendix A.

As mentioned Notary has 2 components: the Server and the Signer. For ease of development and experimentation the Server has the ability to run the Signer inside of the Server process keeping any data in memory that usually would be written to the database. Because we want to simulate a production like environment we will be deploying the Server and the Signer in separate containers within the same Pod [24]. Containers in Pods share the same network namespace which means they can communicate over `localhost`. By binding the Signer to listen only on localhost we ensure that only processes within the Pod can access the Signer. They communicate using mutual TLS or mTLS which enables the Signer to verify if the caller has a valid certificate issued by the same certificate authority (CA). However the Signer does not have the capability to use the common or DNS names inside of that certificate to perform authentication and authorization, it will simply accept any connection from anyone that can present a valid certificate.

They will both store their state in a separate database within the same PostgreSQL [25] instance. Because storage provisioning tends to be handled differently between Kubernetes distributions we've disabled data persistence across restarts. Connectivity with TLS is enforced with authentication happening through certificates and not through passwords. While all connections have TLS enabled, fully securing the database is out of scope. In a production setting you should definitely adhere to the recommended database hardening practises of the chosen database and provide durable storage. This usually includes proper authentication/authorization and process isolation.

A container registry will also be deployed which makes it easier to demonstrate Notary's functionality using our own registry repository names locally. It will also show that even though Docker Hub has a deployment of Notary running, we are not dependent on it to have a fully functioning and trusted registry. The registry we'll be using is the standard `registry:v2` image from the `docker/distribution` repository [26]. Again for the same reasons as before persistent storage is disabled.

When it comes to contacting Notary for inspecting signature data about an image the Docker CLI will default to the same host as is used for the image's registry for example the `registry.example.com` part of `registry.example.com/library/image:tag`. But it has a special case for images pulled from the official Docker registry `docker.io` where it does not use the default behavior and instead switches to use `notary.docker.io` instead [27].

To handle routing HTTP traffic to both Notary and the registry we will make use of an ingress controller [28]. However because of some more complicated rules required to differentiate between requests meant for the registry and requests for Notary we require an ingress controller that can apply pattern matching on URL paths. In our case we've chosen Traefik [29] to handle this for us. It will also handle TLS termination of incoming traffic and re-establishes a TLS connection when contacting Notary and the registry on the backend.

Finally the components require TLS certificates to be provisioned: one set for the Server, one set for the Signer, one set for the registry and one set for PostgreSQL. Instead of manually provisioning them we'll use cert-manager [30] to create a certificate authority for us and

provision TLS certificates. It has a number of certificate issuer types to request certificates from including `Let's Encrypt` and `Vault` [31]. But for our purposes we will use the `CA` issuer which simply requires a key pair which will act as the root CA.

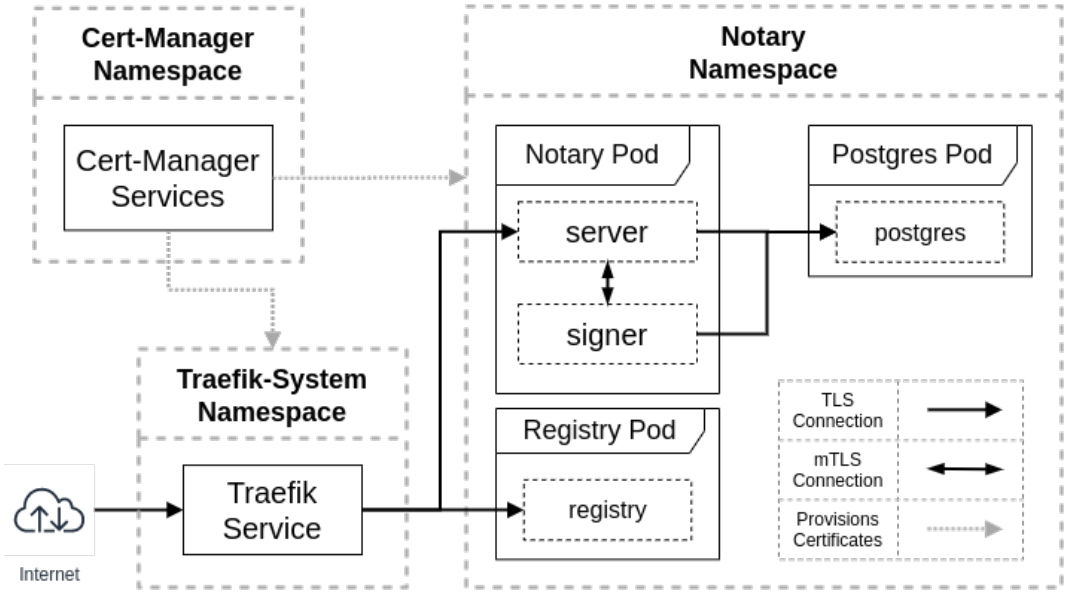

Figure 3: Highlevel Architecture Diagram

5.2 Day 1 - Deployment

We'll now deploy the system using the scripts provided. In broad terms the steps listed below are the general actions performed. A run of the scripts can be found in the `README.md` file of our repository.

1. Download the repository.
2. Create a Kubernetes cluster.
3. Run preflight checks.
4. Deploy Notary, the registry and its dependencies.
5. Validate that we can sign, push and pull images using our deployed services.

5.2.1 Downloading the Repository

Our repository can be found on GitHub[21] and will contain a small number of scripts that help you with downloading the tools, creating the cluster and deploying the services. A guide is offered in `README.md` that will walk you through the happy path in more detail. It also contains all the manifests required for Notary and the registry. You can either browse to GitHub manually and download an archive or use the git command to clone it to your machine. The manifests that you'll find in the deploy folder of the repository include:

- Certificate Resources that will be provisioned and managed by cert-manager.
- A Namespace[32] called *notary*.
- Deployment[33] manifests for the registry including a Service [34] for traffic routing.

- Deployment and Service manifests for Notary with a Job[35] that will download and apply migrations required for both the Server and the Signer.
- IngressRoutes and ServicesTransport custom resources [36] that both help route traffic to the registry and Notary, but also ensures the connections to those are using TLS.

As noted at the top of the `README.md` file in the repository, to maximize the success and repeatability of the tests (not to mention shielding and keeping your machine clean) it is recommended to start a so called Docker-in-Docker setup. It is exactly as it sound, we run a completely separate Docker instance inside of Docker itself. This gives us the ability to easily revert to a clean state while also controlling the test environment. Start a Docker-in-Docker container as shown in listing 1, "exec" into it and install `bash`, `curl` and `git`. From that point on you can clone the repository and start following the guide executing scripts without worrying about modifying your system.

```
$ docker run -d --name dind --privileged docker:dind
0a7eecaacf0876f1c0517c79d4f9cbf48b56442b9a4d95a4334645d5dbfd58c6

$ docker exec -ti dind sh

/ # apk add bash curl git
...snip...

/ # git clone https://github.com/rio/notary-kubernetes.git
...snip...

/ # cd notary-kubernetes
```

Listing 1: Creating a Docker-in-Docker setup.

5.2.2 Create a Kubernetes Cluster

Depending if you already have a cluster available or not you'd want to create one. The script `create-k3d-cluster.sh` will simply start one for you using `k3d` with some minor tweaks to make local ingress work. At the time of writing the bundled version of Traefik in k3s lags behind by a major version, as such we've disabled it and will install our own updated version. We've also instructed the k3d binary to forward ports `80` and `443` to `localhost`. k3s has a feature that when it detects a Service of type `LoadBalancer` it will make that service available on all the nodes of the cluster at the requested ports. This means that we can call `localhost` and it will be forwarded to our deployed Traefik instance.

5.2.3 Preflight Check

For your convenience `preflight-check.sh` will try to find the required tools, check if docker is running and if Kubernetes is reachable. If you decide to use `download-tools.sh` to download the required tools it will install them in the `bin` directory of the cloned repository. By default both the preflight and deploy scripts will look in your `PATH` for the binaries. This means in order to use the binaries downloaded by the `download-tools.sh` script you should add that folder to your `PATH`. The tools used are minimal and typical for working with Kubernetes deployments.

- `kubectl`: The official command line tool [37] to communicate with the cluster and manipulate it's resources. While it is not necessary to use it, as you can call the Kubernetes API directly, it makes it easy to inspect the cluster quickly. We use it to submit our manifests to the Kubernetes API.

- `kustomize`: Is a tool [38] that can transform Kubernetes YAML manifest in a template free way while offering small conveniences like generators and name transformations without touching the original files. We use it mainly as a tool to gather all required manifests and outputting it to kubectl through `stdin`.
- `helm`: Helm, known as "The package manager for Kubernetes" [39] by many, is at it's core a templating tool using the Go templating language [40] to generate Kubernetes YAML files. It also includes facilities to manage the applications lifecycle like installing, upgrading and uninstalling services. Because it is used so much by software packagers to deliver software to Kubernetes we will simply trust that they did a good job. As such we will rely on their knowledge on running both PostgreSQL [41] and Traefik [42] by using the published Helm Charts for those dependencies.

5.2.4 Deploy

Multiple `deploy-*.sh` scripts will install different dependencies and components of the system with `deploy.sh` simply executing them in order. We've split it up to help keep things organized and give you a chance to easily inspect multiple parts of the system if so desired, if not `deploy.sh` will do nicely. It also helps you if you have replacements for certain dependencies deployed or if you need to make small adjustments to them. The scripts generally consist of two types of commands: a command to submit manifests to the cluster and a command to verify that the service is deployed and reporting ready.

Even though the Kubernetes Resource Model [43] and controller model enables the system to eventually reconcile to the desired end state there is no guarantee that every action immediately produces results. This is why the scripts have it's deploy and wait commands structured in a such a way that it should succeed in the first run and is idempotent. One example of when it could fail is if the cert-manager deployment has not had a chance to inform the Kubernetes API of the Custom Resources [44] that it supports before we submit those resources to the cluster resulting in an error where the API simply has no knowledge of the resources. Should it fail it is safe to re-run the script.

It is *not* recommended to run this script against a cluster used for production workloads. A number of cluster wide resources will be created including, `Namespaces`, `Custom Resource Definitions`, `ClusterRoles`, `ClusterRoleBindings`, and both `ValidatingwebhookConfigurations` and `MutatingwebhookConfigurations`. Verify the context and user that you are using by running `preflight-check.sh` and looking for the line containing `kubernetes`.
These resources are created by the dependencies that we use. If you decide that you do not need those dependencies feel free to simply run `deploy-notary.sh`. This will skip the installation of PostgreSQL, Traefik and cert-manager and only install Notary and the registry. Make sure that you adjust the manifests as required if your setup deviates from ours significantly.

With regards to configuring our dependencies we've kept them vanilla with the exception of PostgreSQL. By default the PostgreSQL Helm chart enables persistent storage by creating a PersistentVolumeClaim [45], however we've disabled it simply because not all Kubernetes distro's ship with a mechanism to automatically provision persistent storage. This means that if the PostgreSQL Pod get's killed Notary loses it's state. If your Kubernetes distribution has a storage mechanism (like the `local-path-provisioner` [46] in k3s and kind) feel free to enable it again. TLS is disabled by default so we've also enabled that by pointing to a Secret that is provisioned by cert-manager. The other change we've done to the PostgreSQL helm chart is inject two `.sql` files that, on boot, create the databases and users we require

for the Server and the Signer. Because we've enabled TLS and no password is set by our sql files, PostgreSQL will expect certificates to be presented when clients authenticate. Both the Server and the Signer's database need migrations however the binaries do not apply these themselves but rely on a project called `migrate` [47] to apply them. So along with the deployments we've created a Job using the migrate container image which will clone the Notary repository to a specific tag matching the service versions and apply the migrations.

5.2.5 Validate

Finally we're going to validate that the deployment is functional by stepping through a typical signing workflow. If these commands work we've verified that both the Notary service and the registry are reachable and working properly. To help with that we've included `verify.sh` which is based on Docker's own Content Trust guide [48] and steps through these actions in an idempotent way:

1. Pull in an image.
2. Tag the image so we can push it to our own registry.
3. Generate a role with a private/public key pair on our machine for signing.
4. Add that key as a signer for our image repository. This will generate a new root key and repository key.
5. Sign our image with our private key. This will also trigger a push to the registry.
6. Verify if Docker fails a pull on an unsigned image and succeeds with a signed image.

Each key pair will be encrypted with a passphrase requested on the command line, but for repeatability and ease the script will use a hard coded passphrases that it will print out.

A couple of things to note if you are not running in a sandboxed environment:

- The Docker CLI will try to generate the keys on a Yubikey if it finds one. While it is especially good to have the root keys offline, for this test it might be undesirable or worse, you might overwrite keys on the device. So make sure you unplug it.
- These commands will manipulate files in `$HOME/.docker/trust`. If you already have files in there that you do not want to lose make sure to back them up.

5.3 Day 2 - Operations

Next we will run through some scenarios involving disaster recovery like credential compromise and how to deal with them.

5.3.1 Image Tamper Detection

One of the primary use cases of the Notary project is image tamper detection. This was already part of the image pull pipeline where a layer would be identified and downloaded by it's digest and that digest is then used to validate the integrity of the blob. An example of verification failure can be seen in listing 2.

```
$ docker pull localhost/library/alpine:manipulated-layer
manipulated-layer: Pulling from library/alpine
596ba82af5aa: Verifying Checksum
filesystem layer verification failed for digest sha256:<...snip...>
```

Listing 2: Docker failing the integrity check of a layer.

An image manifest lists the digests for each layer so the client has the ability to verify them individually. However if an attacker manages to modify or flat out replace a layer *and* replace the digest in the manifest, the client will be fooled into accepting the malicious layer. Notary handles this by signing the manifest containing these hashes with a target key. This is in fact what happens if a user has Docker Content Trust enabled and the client pushes a new image. After the layers have been pushed it signs the final manifest and sends it to Notary. When pulling an image, before it even starts downloading any layers, the manifest will be verified by it's sha256 hash and that hash is checked for the validity of it's signature using locally cached public keys. Only if that signature checks out the actual pulling of the layers begins. This acts as a fail fast mechanism so the client is notified almost instantly about problems pertaining the image as seen in listing 3.

```
$ DOCKER_CONTENT_TRUST=1 docker pull localhost/library/alpine:manipulated-
    manifest
Pull (1 of 1): localhost/library/alpine:manipulated-manifest@sha256:<...snip
    ...>
localhost/library/alpine@sha256:<...snip...>: Pulling from library/alpine
manifest verification failed for digest sha256:<...snip...>
```

Listing 3: Docker's Notary integration failing the signature check of a manifest.

When verification failure occurs on either the image layer or manifest an attacker might have modified these files through either the Registry's API or directly on the storage medium. In either case all image layer digests that the manifest referenced are suspect and should archived for investigation before removal from the registry. In the case of `registry:v2` after the deletion of the manifest the `registry garbage-collect` command should be run to remove any lingering blobs of data. Any access logs should also be preserved to assist in identifying the means of access and potentially a source IP address or other identifying information. With regards to restoring the integrity of the registry a trusted build system or engineer should push and sign the entire image.

5.3.2 Target Key Compromise

In Docker Content Trust, the target key determines what tags can be signed and pushed to the repository. Thus, it is also known as the repository key because it determines what tags can be signed into a repository [49]. And following The Update Framework design, a delegation key can be generated to provide multiple trust delegation levels. For Docker Content Trust, there is a unique delegation role known as *targets/releases*. Docker defines the *targets/releases* as the "canonical source of a trusted image tag" [50]. The process of adding a target key to *targets/releases* delegation is automatically done when initializing a repository using the *docker trust signer* command.

If any of the target or delegation keys is compromised, an attacker might sign any malicious image. However, due to the Notary design, an attacker can only tamper within the particular delegation role that the key can sign for and only sign for the specific the type of content allowed. In the unfortunate situation of target key compromise, the first step of mitigating is to revoke the compromised target key from the trust collection immediately. Removing the key as a signer will cause all pull requests to fail for images signed with the compromised key. A new target key should be created, and should be used to sign the trusted collection images again. Resigning the delegation file require the use of the *Notary* client and it is done using the command *notary witness*. A full example of Target key rotation is available in appendix B of this paper.

While The Update Framework proposes using multi-signature trust as a second layer of defense in single or multiple keys compromises. Notary version one only supports a threshold of 1 [51]. This means the images can only be signed by one key, and a key rotation is mandatory in the case of compromise.

5.3.3 Root Key Compromise

Since the root key serves as the trust anchor to all other keys it is particularly sensitive with regards to a compromise scenario. That is why it is recommended to store this offline, only to present it when needed. The Notary project best practices document [52] recommends storing it in a Yubikey *and* have a hard paper copy stored in a vault in case of disaster. If compromise does happen access to the original key is required as it will need to sign the `root.json` document which would contain the new root key's public key thus continuing the chain of trust. Sadly an attacker can do the same along with rotating any other key for the repository. So it is paramount to inspect any key rotations that happened using the old root key and rotate those as well. Root key rotation is more involved than any other type of key rotation and is only possible using the Notary command line tool. A root key rotation flow and a comparison of `root.json` is shown in appendix C.

6 Discussion

Building a reasonably secure deployment on top of Kubernetes proved challenging, given the projects properties and time constraints, but not impossible. During this exercise a number of things surfaced when researching TUF, Notary, the Docker CLI and their interactions which helped inform us when trying to answer our research questions.

6.1 Notary and Image Signing

Notary itself does not sign images but rather facilitates the management of keys, signatures and delegations. It is the client's responsibility, in this case the Docker CLI, to sign any data that they require to be signed and inform Notary. The validation and verification of signatures and image layer integrity is also the responsibility of the client. The Docker CLI performs this job pretty well. However certain constraints put on *what* is actually signed might not fit your use case as it is the manifest that is signed and the image tag that is the identifier used to link the signature to the image being pulled. Not everyone uses tags to identify images but rather directly reference the image digests instead. Also certain tags can be reused a lot, for example `latest` might be reused every time a build is created and `stable` every time something gets released for production use.

In the end it is *not* Notary imposing what does and does not get signed when in normal usage but the client is responsible for what it wants to add to the signature record of a specific repository. As said before the Docker CLI performs adequately and, in most cases, gets out of the way of users even facilitating the use of automation using environment variables to inject decryption passphrases into the signing and validation processes. This is an area where the differences between the Notary and the Docker CLI rear their heads as they both use different environment variables for what looks to be the same purpose, judging by the variable names, but because of slight terminology differences still is not what is expected.

Running Notary yourself usually is combined with a self-hosted container registry, this is where the Docker CLI's usability breaks down. Because of the special case that the CLI has built in for switching to `notary.docker.io` when Docker Hub is detected. For the uninitiated it might seem that the CLI simply contacts Docker's Notary instance through

it's registry domain `docker.io`. The requirement to switch your entire Docker installation to a single Notary instance using the `DOCKER_CONTENT_TRUST_SERVER` environment variable might be undesirable when pulling and pushing to multiple registries. The solution would be to rely on the Docker CLI's fallback behavior which *is* using the registry's hostname. This however reveals that the Notary Server does not have it's own URL path prefix that is usable for routing but instead shares most of the path with the registry only ending with a specific path component as the only means to differentiate between Notary or registry bound traffic. This requires a reverse proxy that can deal with more advanced path parsing to separate a simple `/v2/` prefix from `/v2/any/number/of/path/components/_trust/` path.

6.2 Deployment and Operational Challenges

Deploying Notary on Kubernetes was not as straightforward as we had hoped. While the project's repository does contain a number of Docker Compose setups they do not all have the same level of polish and all will build the project from source. While it is a good starting point to derive manifests from, as the Docker Compose services translate well to Kubernetes Deployments, having a set of solid, officially endorsed and maintained manifests or even a Helm Chart would be a major help. Especially with the existence of tools designed for structurally adjusting manifests for that last change that inevitably has to happen to make it fit your environment. One particularly helpful feature that Notary has built in is the ability to overwrite any part of the configuration file using a predictably structured environment variable. This enables you to swap out or include any sensitive information without having to rewrite the entire configuration file each time. A good example is the password used to encrypt the Signers private keys stored in the database. While environment variables do not come with the same access controls as files do, the ability to selectively control the source of secrets is very valuable when operating a system. Having the ability to load multiple configuration files and merge them would be even better.

While not strictly a problem that Notary causes, Public Key Infrastructure (PKI) [53] is an important part of secure component communication but tends to be hard to setup and manage. In our setup we have delegated it's management to cert-manager which makes it a lot easier to work with, but every organization has its own process for handling it. The split of the Server and the Signer does not make this easier though. When mTLS is not in place, material that has to be signed will be sent in plain text and subject to alteration in flight. When configuring mTLS on the Signers part there is no option to limit connectivity to specific services as they present their certificates, the Signer will simply accept any certificate that originates from the same CA. All of this combined makes the potential of misconfiguration grow and Notary does not make it easy for you to do the right thing. Nor does it help you spot misconfigurations by requiring for example a specific flag to be set acknowledging that you understand that your configuration might be dangerous combining it with a hard to miss warning log message.

Key management and administration can be tricky when people interact with the tools without having a deeper understanding of TUF, Notary, Docker and the relationship between them. Most of your normal day to day interactions will involve the Docker CLI when it comes to signing images, verifying images, generating keys and adding or removing signers from images. But when it comes to key rotation and revocation for example, you will have to interact with the Notary CLI which uses slightly different terminology and exposes more of The Update Framework than the Docker CLI does. These TUF concepts seem to leak out through Notary creating some confusion that can lead to improper use. A user might start with reading about TUF as it is mentioned to be the foundation of Notary but the discrepancies between the two make it hard to bridge concepts when moving from TUF to Notary to Docker. The Docker CLI confuses things more with the specific way it uses delegation roles, making the delegation `targets/releases` mandatory and automatically

rotates the snapshot key into the Signer. Because the Notary system does not enforce these constraints the potential of inadvertently breaking the system for Docker clients exists.

6.3 Probable Reasons for Low Adoption

Deploying and operating Notary safely requires a broad set of skills and a quite thorough understanding of both TUF and Notary concepts. Because of the nature of the project and where it is positioned within the entire delivery and deployment structure, introducing it in an organization could be an involved process. Database administrators, cluster operators, security engineers all need to be involved to build up knowledge around these new concepts and create strict processes around key material management. The Update Framework seems to be well suited for the task of securing software updates with their design goals like minimizing key compromise impact blast radius, but we feel that Notary's abstraction over it leaves much to be desired. Certain concepts of the underlying framework are leaked out to the users creating confusion and the opportunity to make mistakes that could be avoided when using the proper abstraction.

When Notary is in place basic image signing operations should be simple. The developer can enable Docker Content Trust in their container build pipeline and it should take care of signing the manifests, provided the build system has access to the keys of a delegation. This is all assuming that Docker is used to build images, which is far from the only system with that ability and to our knowledge seems is the only one that has Notary support built in. At the deployment end of the pipeline the assumption is also made that Docker is used and has Docker Content Trust enabled. This becomes problematic if Notary isn't deployed along with the internal registry with the same hostname. That Docker instance needs to be informed of the location of the Notary instance residing over that registry which disables that node's ability to pull any image from other registries, like Docker Hub, that do not have signatures in the self-hosted Notary instance. Of course some could consider this a feature. For Kubernetes, some tooling does exist to help validate image signatures before admitting new pod manifests to the cluster but they are few [54] or need a lot of configuration to serve the use-case [55].

Administering Notary is also a challenge with every new system that needs to push requiring their own delegation and key set which in turn requires certain keys to be available for them to be created. Extra tooling is required to make this process easier to manage and maybe even self-serviceable but those tools are also sparse and early in their development [56]. All of these items combined make it quite a hurdle to introduce Notary in an organization. It could be argued that there are other, less involved steps that can be made to get most of the same benefits and confidence in your deployment pipeline. One of which is not relying on image tags in the first place but always reference the image digest directly.

6.4 Notary Version 2

Finally low adoption could also be attributed to the announcement of Notary Version 2. It has been recognised that the initial version of Notary left some things to be desired and warranted taking a step back and reconsider the scope and requirements [57] of the project. Most of the changes proposed revolve around increasing usability, widening scenarios where Notary could be useful and better integration with OCI Artifacts and distribution spec based registries. These changes would most probably improve adoption as it would integrate better in the existing ecosystem and reduce operational complexity. The focus shift to Notary V2 seems to have brought active development on Notary V1 to a halt with the date of the last version released being, at the time of writing, 2018-04-10. It seems like anyone involved and with an interest in container image signing is simply holding their breath.

7 Conclusion

Notary is a great attempt at ensuring the integrity of container images. The primitives it builds on top of from The Update Framework on are quite solid and the main goals have been reached: give developers, operators and systems the ability to sign and validate container images. However, as demonstrated, deploying and implementing Notary Version 1 in a production setup is not straightforward and mostly focuses on Docker based workflows. The service topology and manifests for Kubernetes accompanying this paper attempted to capture good practices and our discussion section adds a number of recommendations around it's day-to-day operations. And while the nature of our main research question "What are the best practices of using Notary for container image signing?" means that it keeps evolving, we think that this is a solid basis to start off with for this version of Notary. Components should be easily swappable and we've catered to it as much as possible within the time constraints.

With most eyes towards the development of Notary Version 2 we expect a lot to change in the coming months. Their requirements set it on a course to improve usability and hopefully increase adoption rate as it should be applicable in more environments and use-cases. For those that cannot wait and have exhausted other possibilities to improve their trust and confidence in the container artifacts they deploy Notary Version 1 should help you achieve that even with some of it's shortcomings as long as you stick to their own recommended practices and general practices regarding distributed systems deployment and operations.

8 Future Work

As of January 2020, no frequent updates are happening to the Notary version 1 project. And the original authors of Notary Version 1's focus shifted toward developing a new version of Notary, also know as Notary Version 2 [57]. Multiple big technology companies are participating in the development efforts of Notary V2. The new project is still at an early stage; However, the latest version will tackle numerous shortfalls of Notary version. Nonetheless, the following research activities can help to add more light to the research questions:

- Perform an external survey, targeting multiple organizations that consume containerized workload in production. The survey's target is to gather more details about the usage and obstacles of adopting container image signing and Notary version 1. Conducting an external survey can add more validation to the conclusion of this paper. However, this task is time-consuming and not performed due to the time constraints of the current project.
- Research the authentication subsystem of Notary and its impact on the security of the container image signing process. At the moment, Notary V1 supports JWT token authentication only between the Client and Notary Server. More analysis is needed to understand the impact or drawbacks of not having authentication between the Signer and the Server.
- Performing analysis for using container image signing using Notary with other frameworks like in-toto [58] to provide holistic security and integrity to software supply chains.

Finally, performing a feedback survey for organizations that will experiment with this paper's manifests will help understand the needed improvements to simplify the usage and accelerate the adoption of Notary and container image signing.

9 Acknowledgments

We thank our supervisors Aristide Bouix and Jasper Boot from KPMG for their time and guidance throughout this research project.

References

[1] SolarWinds. *SolarWinds Security Advisory.* URL: https://www.solarwinds.com/securityadvisory.

[2] *The Update Framework.* URL: https://theupdateframework.io/.

[3] Justin Samuel et al. "Survivable Key Compromise in Software Update Systems". In: Dec. 2010, pp. 61–72. DOI: 10.1145/1866307.1866315.

[4] *Get started with Notary.* URL: https://docs.docker.com/notary/getting_started/.

[5] The Linux Foundation. *CNCF To Host Two Security Projects – Notary and TUF Specification.* URL: https://www.linuxfoundation.org/cloud-containers-virtualization/2017/10/cncf-host-two-security-projects-notary-tuf-specification/.

[6] *The Notary project.* URL: https://github.com/theupdateframework/notary.

[7] Nick Sullivan. *A container identity bootstrapping tool.* URL: https://blog.cloudflare.com/pal-a-container-identity-bootstrapping-tool/.

[8] *Kolide Updater.* URL: https://github.com/kolide/updater/.

[9] Michael Hausenblas. *Results of the 2020 AWS Container Security Survey.* URL: https://aws.amazon.com/blogs/containers/results-of-the-2020-aws-container-security-survey/.

[10] *TUF document formats.* URL: https://github.com/theupdateframework/specification/blob/master/tuf-spec.md.

[11] *PEP 458 – Secure PyPI downloads with signed repository metadata.* URL: https://github.com/theupdateframework/pep-on-pypi-with-tuf.

[12] *JSON Web Tokens.* URL: https://jwt.io/.

[13] *Client-server-signer interaction.* URL: https://docs.docker.com/notary/service_architecture/#example-client-server-signer-interaction.

[14] Stefan Winkle. *Security Assurance of Docker Containers.* Tech. rep. URL: https://www.sans.org/reading-room/whitepapers/cloud/paper/37432.

[15] Tom Ritter, Aleks Kircanski, and Tyler Curtis. *Docker Notary, Application Penetration Test.* Tech. rep. URL: https://www.nccgroup.com/globalassets/our-research/us/public-reports/ncc-docker-notary-audit.pdf.

[16] Dr.-Ing. M. Heiderich et al. *Pentest-Report TUF/Notary 05.-06.2018.* Tech. rep. URL: https://github.com/theupdateframework/notary/blob/master/docs/resources/cure53_tuf_notary_audit_2018_08_07.pdf.

[17] Mason Hemmel and Jeff Dileo. *The Update Framework (TUF) Security Assessment.* Tech. rep. URL: https://www.nccgroup.com/globalassets/our-research/us/public-reports/2017/ncc-group-kolide-the-update-framework-security-assessment.pdf.

[18] Brandon Niemczyk. *Docker Content Trust: What It Is and How It Secures Container Images.* URL: https://www.trendmicro.com/vinfo/hk-en/security/news/virtualization-and-cloud/docker-content-trust-what-it-is-and-how-it-secures-container-images.

[19] *Kubernetes: Production-Grade Container Orchestration.* URL: https://kubernetes.io/.

[20] *Docker Compose.* URL: https://docs.docker.com/compose/.

[21] URL: https://github.com/rio/notary-kubernetes/.

[22] *K3D: K3S on Docker.* URL: https://k3d.io/.

[23] *K3S: Lightweight Kubernetes.* URL: https://k3s.io/.

[24] *Kubernetes Pods.* URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[25] *PostgreSQL: The World's Most Advanced Open Source Relational Database.* URL: https://postgresql.org/.

[26] *Docker Distribution: The Docker toolset to pack, ship, store, and deliver content.* URL: https://github.com/docker/distribution/.

[27] *Docker CLI special case host replacement for Notary.* URL: https://github.com/docker/cli/blob/2291f610ae73533e6e0749d4ef1e360149b1e46b/cli/trust/trust.go#L66-L79.

[28] *Kubernetes Ingress Controllers.* URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

[29] *Traefik Proxy: The Cloud Native Application Proxy.* URL: https://traefik.io/traefik.

[30] *cert-manager: x509 certificate management for Kubernetes.* URL: https://cert-manager.io/.

[31] *cert-manager certificate issuers.* URL: https://cert-manager.io/docs/configuration/#supported-issuer-types.

[32] *Kubernetes Namespaces.* URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[33] *Kubernetes Deployments.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[34] *Kubernetes Services.* URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[35] *Kubernetes Jobs.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/job/.

[36] *The Kubernetes Ingress Controller, The Custom Resource Way.* URL: https://doc.traefik.io/traefik/routing/providers/kubernetes-crd/.

[37] *Introduction to Kubectl.* URL: https://kubectl.docs.kubernetes.io/guides/introduction/kubectl/.

[38] *Introduction to Kustomize.* URL: https://kubectl.docs.kubernetes.io/guides/introduction/kustomize/.

[39] *Helm: The package manager for Kubernetes.* URL: https://helm.sh/.

[40] *Go's text/template package.* URL: https://golang.org/pkg/text/template/.

[41] *PostgreSQL Helm chart packaged by Bitnami.* URL: https://github.com/bitnami/charts/tree/master/bitnami/postgresql.

[42] *Traefik's official Helm chart.* URL: https://github.com/traefik/traefik-helm-chart.

[43] *The Kubernetes Resource Model (KRM).* URL: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture/resource-management.md.

[44] *Extending the Kubernetes API: Custom Resources.*
URL: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[45] *Kubernetes Persistent Volumes.*
URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[46] *Rancher Local Path Provisioner: Dynamically provisioning persistent local storage with Kubernetes.* URL: https://github.com/rancher/local-path-provisioner.

[47] *Migrate: Database migrations written in Go.*
URL: https://github.com/golang-migrate/migrate/.

[48] *Docker Content Trust Guide.*
URL: https://docs.docker.com/engine/security/trust/.

[49] *Manage keys for content trust.*
URL: https://docs.docker.com/engine/security/trust/trust_key_mng/.

[50] *Delegations for content trust.*
URL: https://docs.docker.com/engine/security/trust/trust_delegation/.

[51] *Use the Notary client for advanced users.*
URL: https://docs.docker.com/notary/advanced_usage/.

[52] URL: https://github.com/theupdateframework/notary/blob/master/docs/best_practices.md#root-key.

[53] *Public Key Infrastructure.*
URL: https://en.wikipedia.org/wiki/Public_key_infrastructure.

[54] *Portieris: A Kubernetes Admission Controller for verifying image trust with Notary.*
URL: https://github.com/IBM/portieris/.

[55] *Integrating Open Policy Agent with Kubernetes.* URL: https://www.openpolicyagent.org/docs/latest/kubernetes-introduction/.

[56] *Docker Content Trust - Notary Admin.*
URL: https://github.com/philips-labs/dct-notary-admin.

[57] *Notary V2 Requirements.*
URL: https://github.com/notaryproject/requirements.

[58] *in-toto - A framework to secure the integrity of software supply chains.*
URL: https://in-toto.io/.

Appendices

A Alternative Locally Hosted Kubernetes Options

Multiple other locally runnable Kubernetes distributions exist. A non-exaustive short list follows. Slight adjustments might be required to get our setup to run on these other projects. Generally the differences are around the way that ingress and storage works in these environments.

- **kind**[1]: Similar to k3d/k3s but powered by kubeadm[2].
- **minikube**[3]: Deploys kubernetes in a virtual machine by default but has options for docker and bare metal.
- **Docker Desktop**[4]: Is the go-to way of installing Docker on Mac and Windows platforms and includes a convenient option to turn on kubernetes in it settings.

[1] https://kind.sigs.k8s.io/
[2] https://kubernetes.io/docs/reference/setup-tools/kubeadm/
[3] https://minikube.sigs.k8s.io/
[4] https://www.docker.com/products/docker-desktop/

B Target Key Rotation Example

This is an example for the process of rotating a target key in the case of key compromise. For this example, the image is initially signed with target key *target-a* and it will be replaced with target key *target-b*.

- The image *localhost/library/alpine:signed* is signed using the target key *80d24a5a2986*

```
$  docker trust inspect localhost/library/alpine:signed --pretty

Signatures for localhost/library/alpine:signed

SIGNED TAG   DIGEST       SIGNERS
signed       d0710affa1   80d24a5a2986

List of signers and their keys for localhost/library/alpine:signed

SIGNER         KEYS
80d24a5a2986   b0427237d2d4

Administrative keys for localhost/library/alpine:signed

  Repository Key:       550ba20757
  Root Key:     9f7c261f55
```

- Using the Notary client a list of the available keys can be retrieved from the local repository

```
$ notary key list

ROLE             GUN                          KEY ID         LOCATION
----             ---                          ------         --------
root                                          868a0d87a0     ...snip...
80d24a5a2986                                  b0427237d2     ...snip...
targets          localhost/library/alpine     550ba20757     ...snip...
```

- At this stage *targets/releases* delegation only include *80d24a5a2986* key

```
$ notary delegation list localhost/library/alpine

ROLE                    PATHS             KEY IDS       THRESHOLD
----                    -----             -------       ---------
targets/80d24a5a2986    "" <all paths>    b0427237d2    1
targets/releases        "" <all paths>    b0427237d2    1
```

- In the event of *80d24a5a2986* key compromise the first step is removing the key as signer from the trust collection.

```
$ docker trust signer remove 80d24a5a2986 localhost/library/alpine
Removing signer "80d24a5a2986" from localhost/library/alpine...
The signer "80d24a5a2986" signed the last released version of localhost/
    library/alpine. Removing this signer will make localhost/library/
    alpine unpullable. Are you sure you want to continue? [y/N] y
Enter passphrase for repository key with ID 550ba20:
WARN[0110] role targets/releases has fewer keys than its threshold of 1;
    it will not be usable until keys are added to it
WARN[0110] role targets/releases has fewer keys than its threshold of 1;
    it will not be usable until keys are added to it
Successfully removed 80d24a5a2986 from localhost/library/alpine
```

- Now as the key is removed as signer, any pull request will fail if `DOCKER_CONTENT_TRUST` is enabled:

```
$ export DOCKER_CONTENT_TRUST=1
$ docker pull localhost/library/alpine:signed
WARN[0000] Error getting targets/releases: valid signatures did not meet
    threshold for targets/releases
No valid trust data for signed
```

- And inspecting the image will show that no signature is found:

```
$ docker trust inspect --pretty localhost/library/alpine:signed
WARN[0000] Error getting targets/releases: valid signatures did not meet
    threshold for targets/releases
WARN[0000] Error getting targets/releases: valid signatures did not meet
    threshold for targets/releases
WARN[0000] Error getting targets/releases: valid signatures did not meet
    threshold for targets/releases

No signatures for localhost/library/alpine:signed


Administrative keys for localhost/library/alpine:signed

  Repository Key:       550ba20757
  Root Key:     9f7c261f55
```

- As next step and new key must be generated and added as a signer to the trusted collection.

```
$ docker trust key generate target-b
Generating key for target-b...
Enter passphrase for new target-b key with ID 6d94cd1:
Repeat passphrase for new target-b key with ID 6d94cd1:
Successfully generated and loaded private key. Corresponding public key
    available: /rp1-docker-notary/target-b.pub

$ docker trust signer add target-b localhost/library/alpine --key target-
    b.pub
Adding signer "target-b" to localhost/library/alpine...
WARN[0000] Error getting targets/releases: valid signatures did not meet
    threshold for targets/releases
WARN[0000] Error getting targets/releases: valid signatures did not meet
    threshold for targets/releases
Enter passphrase for repository key with ID 550ba20:
Successfully added signer: target-b to localhost/library/alpine
```

- Now the new *target-b* key is added for the delegation *targets/releases*. However, error for valid signature not meeting the threshold is still thrown.

```
$ notary delegation list localhost/library/alpine
WARN[0000] Error getting targets/releases: valid signatures did not meet
    threshold for targets/releases

ROLE                PATHS             KEY IDS       THRESHOLD
----                -----             -------       ---------
targets/releases    "" <all paths>    6d94cd19b7    1
targets/target-b    "" <all paths>    6d94cd19b7    1
```

- To recover the delegation, the notary witness command is used

```
$ notary witness localhost/library/alpine targets/releases --publish
The following roles were successfully marked for witnessing on the next
    publish:
        - targets/releases
Auto-publishing changes to localhost/library/alpine
WARN[0000] Error getting targets/releases: valid signatures did not meet
    threshold for targets/releases
Enter passphrase for target-b key with ID 6d94cd1:
Successfully published changes for repository localhost/library/alpine
```

- As a final step the image should be signed again with target-b key

```
$ docker trust sign localhost/library/alpine:signed
Signing and pushing trust metadata for localhost/library/alpine:signed
Existing signatures for tag signed digest d0710affa1 from:

Enter passphrase for target-b key with ID 6d94cd1:
Successfully signed localhost/library/alpine:signed
```

- The image is now signed with the new *target-b* key

```
$ docker trust inspect localhost/library/alpine:signed --pretty

Signatures for localhost/library/alpine:signed

SIGNED TAG   DIGEST       SIGNERS
signed       d0710affa1   target-b

List of signers and their keys for localhost/library/alpine:signed

SIGNER     KEYS
target-b   6d94cd19b7d2

Administrative keys for localhost/library/alpine:signed

  Repository Key:       550ba20757
  Root Key:     9f7c261f55
```

- And the image pull is successful again

```
$ docker pull localhost/library/alpine:signed
Pull (1 of 1): localhost/library/alpine:signed@sha256:d0710affa1
localhost/library/alpine@sha256:d0710affa1: Pulling from library/alpine
Digest: sha256:d0710affa1
Status: Image is up to date for localhost/library/alpine@sha256:
    d0710affa1
Tagging localhost/library/alpine@sha256:d0710affa1 as localhost/library/
    alpine:signed
localhost/library/alpine:signed
```

C Root Key Rotation

Here we will demonstrate rotating the root key and the effect it has on `root.json`. Key identifiers have been shortened for better readability. We start out with a root key certificate id `4d7927` for the repository `localhost/libray/alpine` with a `demo-role` delegation added as a signer.

1. First we'll inspect that repository. Note the root key certificate identifier.

```
# docker trust inspect --pretty localhost/library/alpine

No signatures for localhost/library/alpine

List of signers and their keys for localhost/library/alpine

SIGNER      KEYS
demo-role   78d3267110c1

Administrative keys for localhost/library/alpine

  Repository Key:       8ce5d4...
  Root Key:     4d7927...
```

Listing 4: Signature inspection of a repository with the inital root key id.

2. Next we will use the Notary CLI to initiate a root key rotation. This will generate a new key if none is given and use the old key to sign `root.json` that contains the new root key's certificate. This of course requires access to the old root key's material and passphrase.

```
# notary key rotate localhost/library/alpine root
Warning: you are about to rotate your root key.

You must use your old key to sign this root rotation.
Are you sure you want to proceed?  (yes/no)  yes
<... snip password input prompts ...>
Successfully rotated root key for repository localhost/library/alpine
```

Listing 5: Root key rotation command.

3. Running the initial docker trust command again we'll see the new root key's certificate identifier `fcf0a9` for this repository.

```
# docker trust inspect --pretty localhost/library/alpine

No signatures for localhost/library/alpine

List of signers and their keys for localhost/library/alpine

SIGNER      KEYS
demo-role   78d3267110c1

Administrative keys for localhost/library/alpine

  Repository Key:       8ce5d4...
  Root Key:     fcf0a9...
```

Listing 6: Rotated root key.

4. When examining `root.json` we can see a number of things:
   - The document's expiration time has been reset.
   - The new root key's public key has been added to the key list under `signed/keys`.
   - A reference to it under the `signed/roles/root` field has also been added.

- The document's version field has been bumped up.
- A new signature by the new root key is added under `signatures` signifying the validity of this document from the new root key's point of view.
- The signature from the old root key under `signatures` has been updated meaning that the holder of the old key validates the changes to this document.

```
{
     "signed": {
         "_type": "Root",
         "consistent_snapshot": false,
-        "expires": "2031-01-25T19:40:51.488018552Z",
+        "expires": "2031-01-25T19:44:44.931851578Z",
         "keys": {
             "4d7927...": {
                 "keytype": "ecdsa-x509",
                 "keyval": {
                     "private": null,
                     "public": "<...snip public key...>"
                 }
+            },
+            "fcf0a9...": {
+                "keytype": "ecdsa-x509",
+                "keyval": {
+                    "private": null,
+                    "public": "<...snip public key...>"
+                }
             }
         },
         "roles": {
             "root": {
                 "keyids": [
-                    "4d7927..."
+                    "fcf0a9..."
                 ],
                 "threshold": 1
        <... snip ...>
         },
-        "version": 1
+        "version": 2
     },
     "signatures": [
         {
+            "keyid": "fcf0a9...",
+            "method": "ecdsa",
+            "sig": "ob6Ju1..."
+        },
+        {
             "keyid": "4d7927...",
             "method": "ecdsa",
-            "sig": "nWzxZn..."
+            "sig": "KVAepn..."
         }
     ]
 }
```