Harbor :: Scanner Adapters Architecture

Table of Contents

Overview . . . 1
Harbor and Clair . . . 1
Scanner Adapter . . . 3
- Scanner Adapter API . . . 4
- Harbor Configuration . . . 6
- Implementation and Deployment . . . 7
- Summary . . . 9

POC . . . 9
- Microscanner Adapter . . . 9
- Clair Adapter . . . 9

Overview

Currently Harbor is closely coupled with Clair to scan images stored in its registry. The idea is to allow Harbor to work with any other compatible scanner by simple means of configuration change. By doing so Clair could be replaced with any open source or commercial alternative.

The main objective of this proposal was to deliver a minimum valuable product (MVP) of pluggable scanner. At the same time the MVP should be extensible to cater for further improvements.

Harbor and Clair

The current workflow for scanning images is depicted below with the detailed explanation beneath.

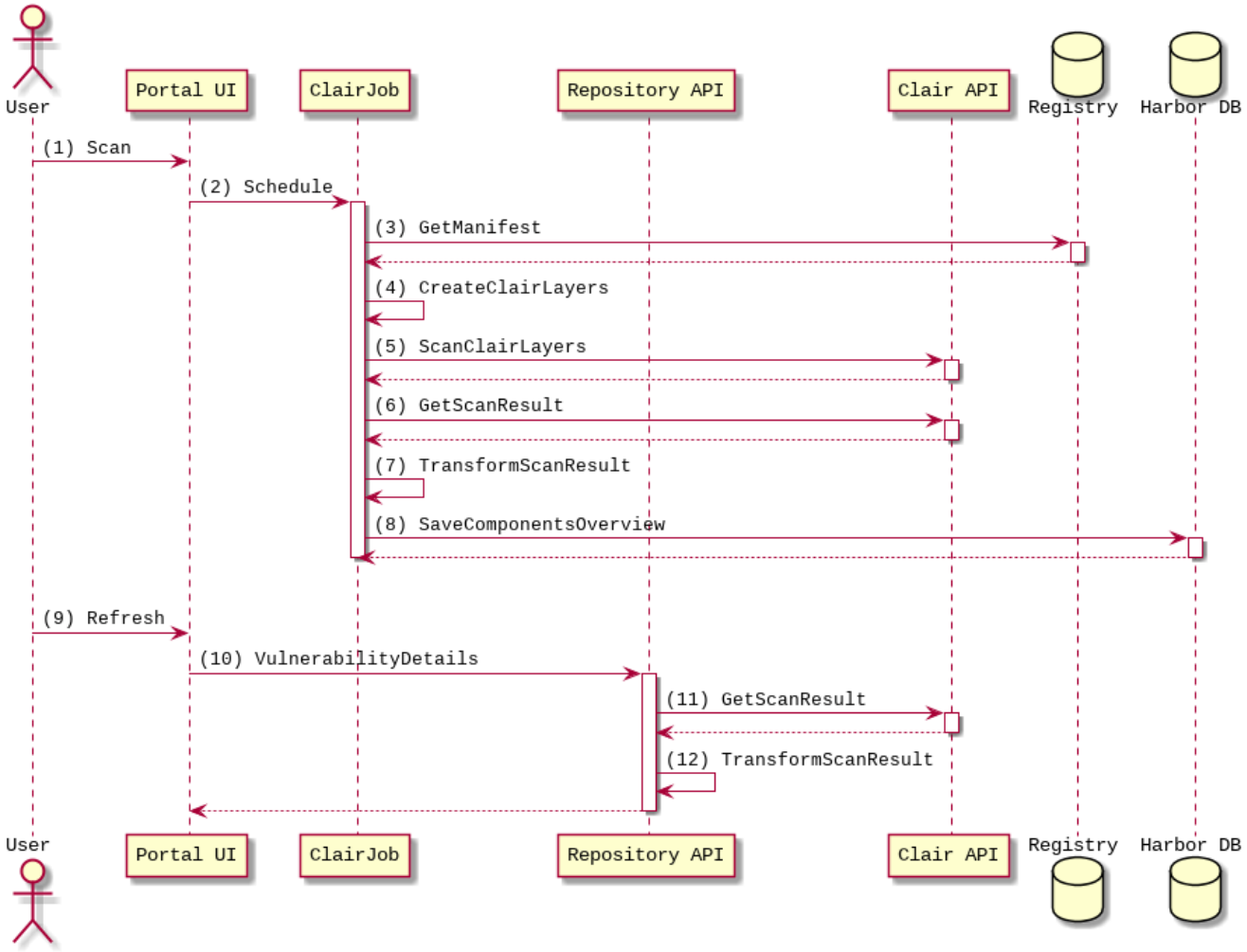

Figure 1. A sequence diagram for the current Harbor integration with Clair.

1. A *User* requests a scan of selected image by clicking the Scan button.
2. The system schedules a *ClairJob* for execution.
3. The *ClairJob* pulls image manifest from *Registry*.
4. The *ClairJob* parses the image manifest. For each image layer it creates an instance of `ClairLayer` structure, which is an internal representation of the layer in Clair. Each `ClairLayer` has `Name`, `Path`, and `Authorization` header properties that allow Clair to pull the image.
5. The *ClairJob* pushes a slice of `ClairLayer` items to *Clair API* for the actual scanning.
6. The *ClairJob* pulls the scan result from *Clair API*.
7. The *ClairJob* transforms scan result to the components overview model represented by the `ComponentsOverview` structure. The `ComponentsOverview` model is understandable by the Harbor's UI and policy checker.
8. The *ClairJob* saves the components overview to the *Harbor DB*. (It's used later on to enforce simple policy rules, e.g. preventing users from pulling an image which contains severe vulnerabilities.) The parent `ClairLayer`s Name is also stored in *Harbor DB* and used later on to fetch scan results.
9. The *User* clicks the Refresh button or the UI timer triggers the scan results refresh.
10. The *Repository API Handler* calls the `VulnerabilitiesDetails` method of *Repository API*.
11. The *Repository API* downloads scan result for the persisted `ClairLayer`'s Name.

12. The ScanResult is transformed to the Harbor's model, i.e. a slice of `VulnerabilityItem`s so it can be rendered in the UI as a grid of vulnerabilities.

Scanner Adapter

"...We can solve any problem by introducing an extra level of indirection."

Having said that imagine that instead of calling directly *Clair API*, the *ClairJob* sends request to *Clair Scanner Adapter API* which in turn calls the *Clair API*. The *Clair Scanner Adapter API* is given all params to pull the image from the Harbor's registry.

This indirection introduced by the *Scanner Adapter API* abstraction allows one to implement a generic `ScannerJob` and `scanner.Client` once and reuse them with every imaginable scanner. In this approach the responsibility of integrating the scanner with Harbor is delegated to the corresponding Adapter's implementation. For example, we can have a dedicated `harbor-clair-adapter`, or `harbor-microscanner-adapter`. The only requirement is that each adapter implements a well defined API and is deployed as a separate microservice.

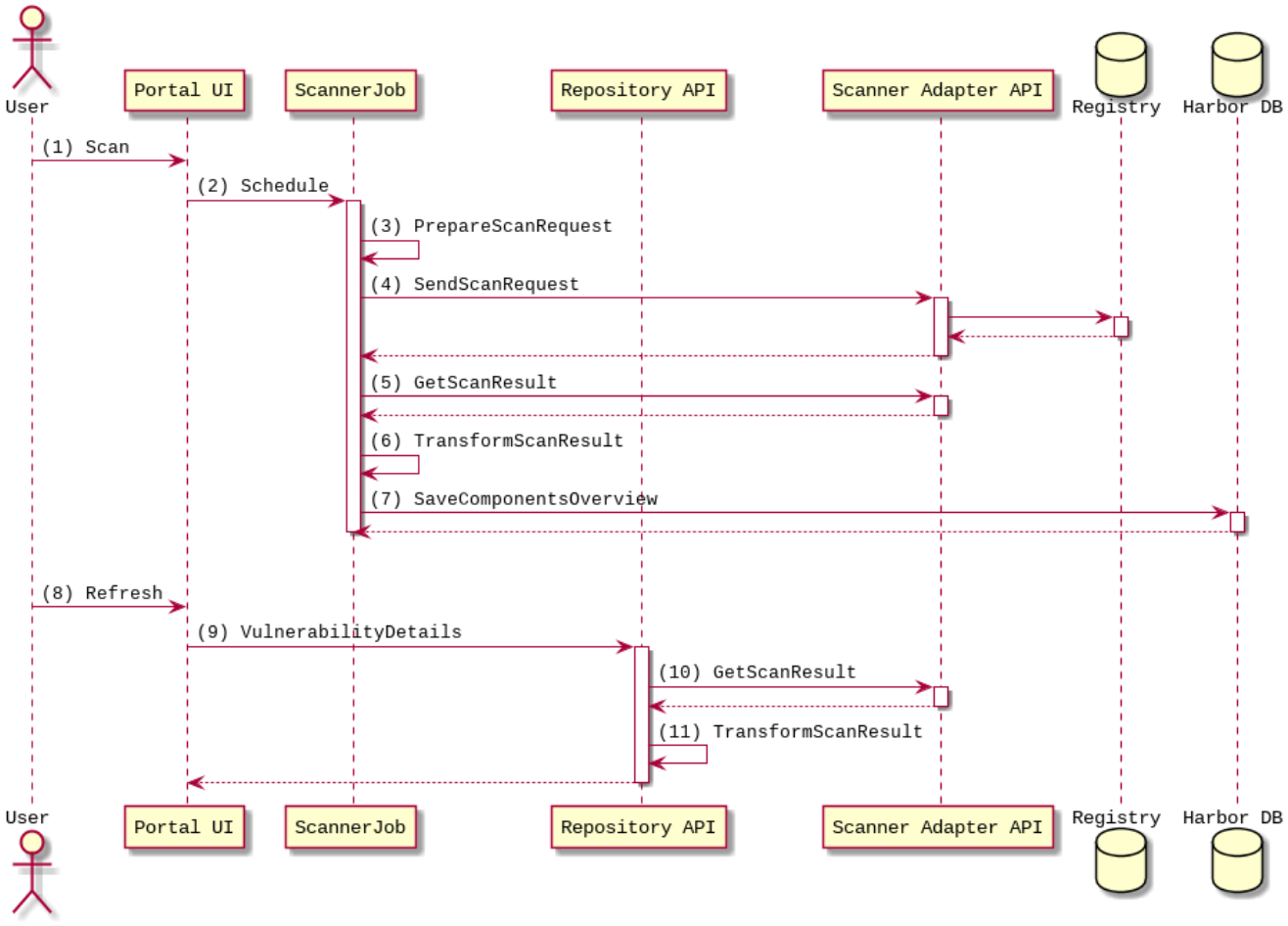

Figure 2. A sequence diagram for the proposed Harbor integration with a Scanner Adapter.

1. A *User* requests a scan of selected image by clicking the Scan button.
2. The system schedules a *ScannerJob* for execution. The *ScannerJob* instantiates the generic `scanner.Client`. The only configuration passed to the client is the endpoint URL of the configured *Scanner Adapter API*.

3. The *ScannerJob* prepares a scan request.
4. The *ScannerJob* submits the scan request to *Scanner Adapter API*.
5. The *ScannerJob* pulls the scan result from *Scanner Adapter API*.
6. The *ScannerJob* transform scan result to the components overview model.
7. The *ScannerJob* saves the components overview to the *Harbor DB*.
8. The *User* clicks the Refresh button or the UI timer triggers the scan results refresh.
9. The *Repository API Handler* calls the `VulnerabilitiesDetails` method of *Repository API*.
10. The *Repository API* pulls scan result for the image digest.
11. The scan result is transformed to the Harbor's model, i.e. the slice of `VulnerabilityItem`s.

Scanner Adapter API

The API is inspired by Clair. It provides the bunch of operations described in the subsequent sections.

API Version Check

A minimal endpoint, mounted at `/v1` will provide version support information based on its response statuses. The request format is as follows:

```
GET /v1
```

If `200 OK` response is returned, the Scanner Adapter implements the V1 API and the client may proceed safely with other operations.

Submit Scan Request

Image scanning is triggered by POST request in the following format:

```
POST /v1/scan
Content-Type: application/json

{
  "registry_url":   "https://harbor-harbor-registry:5000/",
  "registry_token": "JWTTOKENGOESHERE",
  "repository":     "library/oracle/nosql",
  "tag":            "latest"

  "digest: "sha256:9cb763a2a55567ebf4c1d6a70d83d5d032892c8d5aee8ea5894ef0a3c3786e54",
}

{
  "details_key":
"sha256:9cb763a2a55567ebf4c1d6a70d83d5d032892c8d5aee8ea5894ef0a3c3786e54" ①
}
```

① The details key is used to fetch scan result. It can be digest or any other identifier. For example, Clair is using layer name as a key to fetch scan result.

If `201 Accepted` response is returned, the scan request was submitted successfully and the client may proceed with getting the corresponding scan result.

The JSON request payload contains all data that allows Scanner Adapter to pull image from the Harbor's Registry. For example, it should be able to send the following requests:

```
GET https://harbor-harbor-
registry:5000/v2/library/oracle/nosql/manifests/sha256:b1165286043f2745f45ea637873d619
39bff6d9a59f76539d6228abf79f87774
Authorization: Bearer JWTTOKENGOESHERE
```

```
GET https://harbor-harbor-
registry:5000/v2/library/oracle/nosql/blobs/sha256:b113c8b260349e1adcfea8f2909d26e4a0a
5c3bb6ef6e93e47fc22cf8d3fc7d5
Authorization: Bearer JWTTOKENGOESHERE
```

Get Scan Result

To get the scan result for the given image digest the following request has to be sent:

```
GET /v1/scan/<detailsKey>

{
  "overview": {
    "total": 2,
    "summary": [
      {"severity": 1, "count": 0},
      {"severity": 2, "count": 0},
      {"severity": 3, "count": 1},
      {"severity": 4, "count": 0},
      {"severity": 5, "count": 1}
    ]
  },
  "vulnerabilities": [
    {
      "id": "CVE-2017-18018",
      "severity": 5,
      "package": "coreutils",
      "version": "8.23-4",
      "description": "In GNU Coreutils through 8.29, chown-core.c in ..." chown and
chgrp does not prevent replacement of a plain file with a symlink during use of the
POSIX \"-R -L\" options, which allows local users to modify the ownership of arbitrary
files by leveraging a race condition.",
      "link": "https://security-tracker.debian.org/tracker/CVE-2017-18018"
    },
    {
      "id": "CVE-2017-8283",
      "severity": 3,
      "package": "dpkg",
      "version": "1.17.27",
      "description": "dpkg-source in dpkg 1.3.0 through 1.18.23 is able to use a non-
GNU patch program and does not offer a protection mechanism for blank-indented diff
hunks, which allows remote attackers to conduct directory traversal attacks via a
crafted Debian source package, as demonstrated by use of dpkg-source on NetBSD.",
      "link": "https://security-tracker.debian.org/tracker/CVE-2017-8283"
    }
  ]
}
```

The returned JSON which represents scan results reuses the current Harbor's model for components overview (`ComponentsOverview`) and vulnerability representation (`VulnerabilityItem`). This is done deliberately in V1 of the API to minimize the impact of changes in the code (JavaScript / DB migrations) but still deliver a MVP.

Harbor Configuration

The Harbor's config would have a very generic structure as the only required config param is the

URL of the *Scanner Adapter API*. In other words, Harbor is not aware of any vendor specific configuration options such as access tokens, upstream vulnerability databases and so on. Vendor specific scanner configuration should be handled by the scanner's adapter and the scanner itself.

A snippet of Harbor config pertinent to the image scanning.

```
# You can switch an image scanner by changing its endpoint URL.
imageScanner:
  # Use CoreOS Clair for image scanning
  name: "Clair"
  vendor: "CoreOS"
  endpointURL: "http://harbor-clair-adapter:6000/"

  # Alternatively use Aqua Security Microscanner
  # name: "Microscanner"
  # vendor: "Aqua Security"
  # endpointURL: "http://harbor-microscanner-adapter:8080/"

# See https://martinfowler.com/articles/feature-toggles.html
featureToggles:
  # If it's turned on a new scanner adapter is enabled, if it's off we
  # fall back to the existing scanning with Clair.
  SCANNER_ADAPTER: "on" ①
```

① A very simplistic approach to implement a feature flag mechanism.

Implementation and Deployment

The implementation of such architecture can be executed as follows:

1. Introduce a feature toggle, e.g. `SCANNER_ADAPTER=[on|off]`, to enabled/disable scanner adapters functionality. This will allow us to experiment and deliver the code incrementally.
2. Implement a generic `scanner.Client` to communicate with the *Scanner Adapter API*:

```
package scanner

// ScanRequest represents a structure that is sent to Scanner Adapter API
// with all the details required to fetch image meta-data and layers.
type ScanRequest struct {
    RegistryURL   string `json:"registry_url"`
    RegistryToken string `json:"registry_token"`
    Repository    string `json:"repository"`
    Tag           string `json:"tag"`
    Digest        string `json:"digest"`
}

type ScanResponse struct {
    DetailsKey string `json:"details_key"`
}

// ScanResponse represents the outcome of the image scan.
type ScanResult struct {
    Severity Severity `json:"severity"`
    Overview *ComponentsOverview `json:"overview"`
    Vulnerabilities []*VulnerabilityItem `json:"vulnerabilities"`
}

// Severity represents the severity of a image/component in terms of vulnerability.
type Severity int64

type ComponentsOverview struct {
    Total   int `json:"total"`
    Summary []ComponentsOverviewEntry `json:"summary"`
}

type ComponentsOverviewEntry struct {
    Sev int `json:"severity"`
    Count int `json:"count"`
}

type VulnerabilityItem struct {
    ID          string   `json:"id"`
    Severity    string   `json:"severity"`
    Pkg         string   `json:"package"`
    Version     string   `json:"version"`
    Description string   `json:"description"`
    Link        string   `json:"link"`
    Fixed       string   `json:"fixedVersion,omitempty"`
}

type ImageScanner interface {
    Scan(req ScanRequest) (*ScanResponse, error)
    GetResult(detailsKey string) (*ScanResult, error)
}
```

3. Implement `ScannerJob` by porting the logic from `ClairJob` and using a fresh `scanner.Client` instead of existing `clair.Client`.
4. Modify the code that actually schedules `ClairJob`. The code should read the `SCANNER_ADAPTER` feature flag. If it's `off` it should fallback to submitting a `ClairJob`. If it's `on` it should run the `ScannerJob`.
5. Similarly modify the *Repository API* HTTP handler for fetching scan details, i.e. if the `SCANNER_ADAPTER` feature flag is `on`, use `scanner.Client` instead of `clair.Client`.
6. Implement `clair-harbor-adapter` as a reference implementation. Host it in a dedicated repository, e.g. https://github.com/goharbor/harbor-clair-adapter.

Summary

Advantages

1. Quite simple to implement incrementally and deploy behind a feature toggle.
2. Preserve existing data model. No changes to the database models.
3. Scalable in terms of Harbor's code base and community contributions. Not a monolith. (Harbor does have to know about Scanner X or Scanner Y. Instead Scanner X and Scanner Y knows about Harbor.)
4. DRY Write `scanner.Client` once and reuse it everywhere.

Disadvantages

1. Additional abstraction layer and additional hop in troubleshooting problems or debugging code.
2. Maintain the scanner adapter's API.
3. Evaluate upfront whether the API is flexible enough to cater for all use cases.

POC

https://github.com/danielpacak/harbor/tree/scanner_adapters_poc

Microscanner Adapter

https://github.com/danielpacak/harbor-microscanner-adapter

Clair Adapter

https://github.com/danielpacak/harbor-clair-adapter