Proposed Scanning Workflow

STATUS: DRAFT (PUBLIC)

OWNER: david.lawrence@docker.com

Background

Security automation is a critical tool in software development. Scanning in a variety of forms is a significant addition to a Continuous Integration (CI) pipeline that greatly reduces both the maintenance burden and opportunity for human error.

Being such a useful component, it should be as straightforward as possible for everyone to integrate any array of scanners into their workflow. To enable this, the Scanning SIG has been formed to work on standardizing the format of vulnerability reporting to enable CI systems and artifact storage providers to provide broad support for any scanners their users wish to integrate with.

Overview

There are 3 core components to the scanning workflow:

1. Generating the Bill of Materials (BoM) for an image
2. Analyzing the BoM and files themselves for vulnerabilities
3. Attaching the resultant report to the image

The goals of this SIG are similarly threefold:

1. Standardize the Bill of Materials format
2. Standardize the scan report format
3. Standardize how to attach a report to an image in a trusted manner

The core benefits to scanning providers are a great simplification to potential customers in integrating a scanning service into their workflow. Following Docker's "batteries included but pluggable" philosophy, it should be easy for users to select the scanning service of their preference without having to build custom integrations. This will allow scanning services that implement the standardized formats to provide a native looking and feeling integration with Docker's hosted and on-premise solutions.

To meet the first two goals, this document presents a strawman and looks to the community represented in this Scanning SIG to move the discussion forward. The scanning SIG will discuss the proposal and improve the design to create a flexible and robust solution.

Finally this document presents a proposal that incorporates the Notary framework, an implementation of The Update Framework (TUF), to provide cryptographic trust tying a report to an image. Notary is used by Docker and other registry service providers, and soon to be submitted to the Cloud Native Computing Foundation (CNCF).

Details

The proposed approach to scanning is a 2-step solution:

1. Generate a *bill of materials* that contains a list of all the discovered *components and their versions* for each layer of a Docker image
    a. The BOM changes on an infrequent basis, mostly when a new version of a scanner is available that can detect new components
    b. For each layer in the Dockerfile, generate a *BOM*
    c. This may be a fairly resource intensive operation and take tens of minutes (depending on the size of the image)
    d. Things other than components (ie SSH or AWS keys baked in, etc) may be surfaced here as well
2. Run the BOM through a vulnerability database to get the latest vulns found for each known component
    a. This should be a fairly light and fast operation
    b. The underlying vuln DB will to be updated on a regular basis to keep up with new CVE discoveries. This step will need to be re-run periodically.

The report having been generated, it will be attached to the image via the scanning service signing a TUF delegation file, tying the image checksum to a deterministic ID for the report. The use of a deterministic ID is important in enabling a user or other consuming system to assess the correctness and integrity of the report they are inspecting for a given image.

BOM

The BOM should contain the following information:

- An integer schema version

- Namespace/reponame/tag of the image
- Whether or not the image has *foreign* layers (skipped since the bits reside somewhere else)
- For each layer:
    - SHA256 of the layer (comes from Docker image *manifest*
    - Layer size
    - Docker build command line
    - List of components
- For each identified component:
    - Name
    - Version
    - License (ie GPL, MIT, Apache, etc)
        - URL to that license for users to easily find it
    - Path to file(s) on disk where that component is present in this layer

BOM Example:

```
{
   “version”: “1”,
   "image": "sha256:4bbbc93d57...",
   "layer_details":[
      {
         "sha256sum":"10a267c67f4...",
         "size":52584016,
         "docker_command_line":"/bin/sh -c #(nop) ADD
file:f4e6551ac34ab446a297849489a5693d67a7e76c9cb9ed9346d823
92c9d9a5fe in / ",
         "components":[
            {
               "component":"acl",
               "version":"2.2.52-2",
               "license":{
                  "name":"LGPL",
                  "type":"lgpl",

"url":"https://www.gnu.org/licenses/lgpl.html"
               },
               "fullpath":[
                  "/lib/x86_64-linux-gnu/libacl.so.1.1.0"
               ],
```

```
    "vulns": "<defined below>"
}
```

Vulnerability Check

Ultimately, the *BOM* is sent through the “fill vulnerability list” to receive a decorated output containing all the vulnerabilities found for each known component. The format of the report must support multiple types of scanning, for example, binary scanning for CVEs and SQLMap scanning for SQL injection attacks.

The following data should be included in the output:

- A vulnerability type, e.g. “CVE”, “SQLi”
- A reference ID where appropriate, i.e. CVE-2016-7543
    - Vulnerabilities that are not part of a database such as mitre may omit this field.
- A score.
    - For CVE types, this should be the CVSS.
    - For other types, this should be a severity score on a 1-10 point scale.
- Summary, an english string describing the vuln
- Any notes for the vulns
    - If the upstream has determined the vuln to be *demoted* or *wontfix*
    - If the vuln is *ignored* by upstream
-

Sample JSON response for vulnerability check:

```
"vulns": [
    {
        "type": "CVE",
        "id": "CVE-2016-7543",
        "score": 7.2,
        "summary": "Bash before 4.4 allows local users to
     execute arbitrary commands with root privileges via crafted
     SHELLOPTS and PS4 environment variables.",
        "notes": null,
        "remediation": {
          "url": string,
          "fixed_version": string,
          "notes": [
            "Upgrade"
            "Change configuration to xyz"
          ]
```

```
    }
          “    {
        “type”: “sqli”,
        "score": 9,
        "summary": "popd in bash might allow local users to
     bypass the restricted shell and cause a use-after-free via
     a crafted address."
        "notes": null
    },
    ...
]
```

Attachment of report to an image

Docker images are currently signed using Notary, with the checksum of the image manifest, and the human readable label for the image, being signed together into the trust data. The TUF specification permits each of these records to include a “custom” field. The only requirement on this field is that it is a valid JSON object, and it may otherwise contain any data the user deems useful.

For performance purposes it is beneficial to keep the data included small, and therefore only to include data that clearly indicates the type of data, and a deterministic ID. The simplest example assumes the reports themselves are stored in the same registry as the image, and therefore no explicit location data is required. A sample custom field might look like:

```
{
    “report”: {
        “type”: “scanner”,
        “id”: {
            “sha256”: “c39b2a7a850b...”
        }
    }
}
```

A given scanning service will have its own file into which its own reports will be signed so there is no danger of collisions if a user attaches multiple scanners. Notary also maintains historical versions of the signed file so there is no need for multiple versions of a report to be attached to an image, any auditing can be done by inspecting the historical Notary data.

Signing alone provides users with strong guarantees as to the origin and authenticity of the report. Signing within the context of Notary and TUF, also provides strong guarantees that the report is current, as TUF’s specification is specifically designed to protect against freeze and

rollback attacks. Finally, attaching a deterministic ID for the report to the deterministic ID of the image provides strong guarantees on the integrity of the report and the integrity of its association with a specific image.