UBCIS: Ultimate Benchmark for Container Image Scanning

Shay Berkovich
BlackBerry Limited

Jeffrey Kam
University of Waterloo

Glenn Wurster
BlackBerry Limited

Abstract

Containers are regularly used in modern cloud-native deployment practices. They support agile and continuous integration/continuous deployment (CI/CD) paradigms, isolating services. As containers become more ubiquitous, container security becomes crucial as well. Scanning container images for known vulnerabilities caused by vulnerable software is a critical security activity of the CI/CD process. Both commercial and open-source tools exist for container image scanning. Results from these scanners, however, are inconsistent. Inconsistent results make it hard for developers to choose the best solution for their environment. In this paper, we present the Ultimate Benchmark for Container Image Scanning (UBCIS), a benchmark for evaluating image scanners. UBCIS contains a classification of known vulnerabilities in common base container images, as well as a framework for running container vulnerability scanning tools. UBCIS makes it possible to evaluate scanners. We discuss intricacies of classifying vulnerabilities, presenting a process that can be used when determining the relevance of vulnerability. Finally, we provide recommendations for choosing the best scanner for a specific environment.

1 Introduction

Many container image vulnerability scanning tools are being introduced by commercial and open-source groups. Container image vulnerability scanning is focused on finding instances of already-known vulnerabilities in binaries on the system. This is in contrast to many vulnerability scanning tools (e.g., static analysis), which are focused on finding new or undiscovered vulnerabilities. This is also in contrast to detecting malicious container interactions [23].

Docker encourages container image reuse by making it easy to obtain base images and extend them. There are multiple registries hosting public images, including Docker Hub [8], GCR [11], and Quay [21]. However, the same reasons that facilitate rapid container adoption also increase the risk of using vulnerable software. Zerouali et. al. [29] suggest over half of the images hosted on Docker Hub have not been updated in four months or more, and that one out of every five installed packages in a container is outdated. Shu et. al. [22] document no significant difference between community and official images. The same study shows that using `latest` images [18] does not eliminate the need for scanning. Using a tool to scan for known vulnerabilities in your image is therefore critical to the security of the system.

The Ultimate Benchmark for Container Image Scanning (UBCIS) is designed to evaluate the precision, recall, and F-measure of container image vulnerability scanning tools. We address two main problems with this work. The first is the classification of vulnerabilities detected by scanning tools. Our work subdivides container image vulnerabilities based on their applicability to the container image. The second problem addressed is providing a framework that can be used to assess new scanning tools. We measure the ability of a scanner to detect vulnerabilities caused by out-of-date applications.

Our contributions are 1) a benchmark tool for container scanner evaluation; 2) an evaluation of three popular scanners on common container images; 3) a vulnerability judging process for classifying vulnerabilities; 4) a set of vulnerabilities which have been judged and can be used with the benchmark tool to evaluate scanners; 5) recommendations for choosing a scanner; and 6) an in-depth analysis of how scanners interpret different vulnerability classes and how that interpretation affects the precision, recall, and F-measure of the scanner. This benchmark has solved the problem of choosing the best scanner for our production container deployments.

In addition to being used within corporate environments to choose the right container scanning tool, UBCIS can also be used in studies that require container image scanning. Current studies use specific open-source scanners as a single source of truth on the number and type of vulnerabilities in the container image [14,15,22,24]. As we show, results vary between scanners, potentially affecting studies suppositions. UBCIS will empower scholars to choose the appropriate image vulnerability scanner when engaging in related research.

In Section 2, we discuss applicability classes for vulnerabilities a scanner detects. We create a benchmark and evaluate three scanners in Section 3. Section 4 documents our observations and recommendations. Section 5 contains related work. We conclude in Section 6.

2 Classification of Vulnerabilities

Scanning for known vulnerabilities in a container image involves three steps: (1) identifying all components (e.g., executables, libraries, scripts) of the image, along with their version; (2) given the list of components, querying security feeds for applicable vulnerabilities; and (3) reporting each vulnerability affecting a component in the container image. The following list highlights why different scanners may give different results:

- Most scanners query the package manager, while few perform binary analysis. The two approaches might cause different versions to be detected.
- While common vulnerabilities can be found in a single vulnerability feed such as National Vulnerability Database (NVD) [20], most scanners employ a set of vulnerability feeds (including some commercial) to use as many sources of potential vulnerabilities as possible. The list of feeds, along with the feed prioritization, differs greatly from scanner to scanner.
- Some scanners authors curate vulnerability feeds, leading to a lag in time between the vulnerability being known and being reported by a specific scanner.
- Ambiguity - It is sometimes unclear whether a vulnerability exists in an open source component, or if it is present in the component as deployed in the container. Scanners authors can weight vulnerability feed information differently when deciding whether to report a vulnerability in a container.

Consider `debian:10.2`, a popular and widely-adopted base image. Running four different scanners on this image results in four different sets of vulnerabilities. No set is a superset or a subset of another, no set encompasses all image vulnerabilities, and every set contains at least one false positive. Customers looking to procure an image scanner may not choose the best tool for their environment if they merely look at the number of detected vulnerabilities without considering other factors.

2.1 Applicability Classes

To quantify the impact of scanner design choices, we ran four different scanners on three different container images, manually examining each detected vulnerability. Based on this analysis, we have identified several applicability classes for vulnerabilities detected by a scanning tool, expanding on GitHub [12]. We call classes (*I*, *MM*, and *D*) ambiguous classes.

TP / True Positive - Vulnerability is present in the container.

I / Inconclusive – It is not clear whether the vulnerability is present. There might be insufficient information to confirm the presence of the vulnerability. Newly discovered vulnerabilities that have not yet been examined and added to the UBCIS database fall into this class.

MM / Version Mismatch – Vulnerability where different feeds disagree on fixed or affected package versions (e.g., NVD lists CVE-2018-12886 [5] as affecting version 4.1 through version 8. It is unclear whether 8.8 would be within this range. In our testing, 50% of scanners reported the issue).

D / Disputed – Vulnerability that is disputed by maintainers (e.g., CVE-2019-9192 [6]).

FP / False Positive – Vulnerability is not applicable to the container image. This can be due to differences in packaging for the distribution, or back-ported fixes.

3 Scanner Evaluation

The most obvious scanner evaluation metric is how many vulnerabilities are reported. This metric in isolation is error prone. A better metric of the scanner success is the *Relevant* vulnerabilities detected by the scanner under evaluation as true positives (`TP`), with *not-relevant* vulnerabilities being false positives (`FP`). *Relevant* vulnerabilities not detected by the scanner are false negatives (`FN`). Precision is defined as the fraction of retrieved vulnerabilities that are in fact relevant, or $Precision = TP/(TP+FP)$. Recall is the fraction of relevant vulnerabilities detected by the scanner, or $Recall = TP/(TP+FN)$. The F-measure characterizes the combined performance of recall and precision, or $F\text{-}measure = (2*Recall*Precision)/(Recall+Precision)$. We use these three metrics to assess scanner quality.

3.1 Docker Image Choice

Debian, Alpine and Ubuntu make up over 87% of docker base images on Docker Hub [24]. CentOS, Buildroot and Fedora lag significantly. Image pull numbers confirm the popularity of Debian, Alpine, and Ubuntu [15, 24]. We use these three distributions in our evaluation, supporting them in UBCIS.

To choose a specific image, we need to choose a tag. Within the registry, every container image can be uniquely described by the tuple `repo-name:tag`, where `repo-name` is the name of the image and `tag` is the version. For OS-level base images, *repo-name* is always the distribution and `tag` is usually the distribution version (e.g., `debian:buster`, `ubuntu:18.04`, or `alpine:3.10`). The `latest` tag denotes the most recent rolling image version. We choose images in Table 1 that are stable (not `latest`), but also popular, being used in current deployments so that the scanner evaluation is relevant.

Another good reason to choose popular images is to prevent scanners from gaming the system. Any scanner that scores well in the benchmark tests will, by definition, score well on the majority of real-world images. By benchmarking scanners, we encourage continual improvement on real-world data sets.

Image	Repo Pulls	Last Update
`debian:10.2`	100M	February 3, 2020
`alpine:3.9.4`	1B	June 19, 2019
`ubuntu:18.10`	1B	July 23, 2019

Table 1: Images used for the benchmark

While a stable image will not change, the list of vulnerabilities found against this image will change over time. Newly discovered vulnerabilities will impact the stable image, requiring periodic benchmark regeneration. We use Vagrant [28] to automate the process.

3.2 Process

To build the benchmark, we merged the findings of multiple scanners; Anchore [2], Trivy [26], Clair [4], and a binary scanner. Anchore, Trivy, and Clair all use the container package manager to obtain a list of installed software. The binary scanner attempts to detect binaries and their version numbers without using the package manager. Different component retrieval techniques ensure better coverage of detected packages and thus better coverage of discovered vulnerabilities.

We ran the scanners in their default configuration, ensuring feeds are available. For each vulnerability reported by each scanner in each image, we manually judge the vulnerability to determine its applicability class (see Section 2.1), generating a list of all vulnerabilities found by any scanner. Overall, we judged 146 vulnerabilities for Debian, Alpine, and Ubuntu images.

We call the process of classifying the reason for the detected vulnerability the vulnerability judging process. This process is manual and non-trivial. We perform the following sequence of steps, in order, until we have a result:

- D0 Determine the package name, version, and metadata.
- D1 Is the vulnerability already triaged? If so, use the result.
- D2 Is the vulnerability language or distribution specific? We ignore vulnerabilities in language specific package repositories such as NPM, PIP, or Ruby Gems at this point in time.
- D3 Is the vulnerable package detected by the scanner empty? If so, mark as a false positive.
- D4 Is the vulnerability applicable to the distribution? If not, mark as a false positive.
- D5 Is the vulnerability fixed in the distribution? If so, mark as false positive. If not, mark as a true positive.

Step D3 is interesting as some scanners flag a meta-package (i.e., a package without content) as containing a vulnerability. This vulnerability is a False Positive as there is no code that can be vulnerable (e.g., `libc-utils` in Alpine flagged by some scanners is an empty/meta package [17]). Step D4 is a common source of variance between scanners. Reasons range from wrong operating system (e.g., a Windows vulnerability in Linux-based distros), to specific distro-level package usage that changes the applicability of the vulnerability. The relevant information in a vulnerability feed does not follow a standard format. It can appear in multiple places, causing discrepancies. D5 is another source of variance between scanners because they fail to recognize backports and ad-hoc fixes, leading to false positives. If questions raised in D4 and D5 cannot be answered, the vulnerability will fall into an ambiguous class (Section 2.1).

We choose to put more weight in distribution-specific security feeds (i.e., Debian Security Tracker [7], Alpine-secdb [1], or Ubuntu CVE Tracker [27]) than in general security feeds like NVD [20]. Distribution-specific feed maintainers have more information and expertise to determine what is applicable in their case.

3.3 Benchmark Modes

To generate precision, recall, and F-measure metrics, all ambiguous classes must be mapped to either false positives or false negatives. We define two modes of benchmark evaluation: paranoid and relaxed. Paranoid mode maps ambiguous classes to true positives. Relaxed mode maps ambiguous classes to false positives. Choosing paranoid or relaxed mode when evaluating scanner results will depend on the risk tolerance of the company using the scanner, impacting the benchmark evaluation and result.

		Trivy	Anchore	Clair
Debian 10.2	True Positives	42	22	37
	Inconclusive	6	7	7
	Mismatch	4	1	4
	Disputed	2	2	2
	False Positives	0	0	2
	Total	54	32	52
Alpine 3.9.4	True Positives	5	5	1
	Inconclusive	0	0	0
	Mismatch	0	0	0
	Disputed	0	0	0
	False Positives	0	0	0
	Total	5	5	1
Ubuntu 18.10	True Positives	0	11	10
	Inconclusive	0	5	4
	Mismatch	0	6	5
	Disputed	0	0	0
	False Positives	0	2	2
	Total	0	24	21

Table 2: Vulnerability totals and groups per scanner/image.

4 Observations

Table 2 shows the results of three scanners. The total number of detected unique vulnerabilities differ wildly, highlighting

		Trivy			Anchore			Clair		
		Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
Debian 10.2	Relaxed	0.78	0.98	0.87	0.69	0.51	0.59	0.71	0.86	0.78
	Paranoid	1.00	0.69	0.82	1.00	0.41	0.58	0.96	0.64	0.77
Alpine 3.9.4	Relaxed	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.20	0.33
	Paranoid	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.20	0.33
Ubuntu 18.10	Relaxed	NA	0.00	NA	0.46	0.73	0.56	0.48	0.67	0.56
	Paranoid	NA	0.00	NA	0.92	0.42	0.57	0.90	0.36	0.51

Table 3: Precision, Recall and F-measure per scanner per benchmark mode.

the impact of choices discussed in Section 2. Furthermore, unsupported images are a real problem. Although we were planning on including Fedora, we discovered during testing that most scanners did not support it. Also, Trivy does not support Ubuntu 18.10.

Many vulnerabilities on `debian:10.2` and `ubuntu:18.10` fall into ambiguous classes based on manual analysis. The high number of vulnerabilities falling into ambiguous classes in Table 2 highlights the significance of relaxed and paranoid mode in Table 3 . With only two options available (*report* or *ignore*), scanners address inconclusive results by expanding vulnerability severity scale to include values such as *Unknown*, *Negligible* or *Unimportant*. By including inconclusive results, the scanner defers analysis to the customer.

Table 3 shows that the best scanner for Debian would be Trivy in both relaxed and paranoid mode. For Alpine, Anchore and Trivy are equally good. For Ubuntu, Anchore and Clair show similar results.

No scanner is best for all combinations of images and benchmark modes. Companies exploring the purchase of an image scanner should follow these recommendations:

1. Assess risk tolerance. Can we afford to miss vulnerabilities (relaxed mode), or must we treat all vulnerabilities as potentially critical (paranoid mode)? Scanners with a better paranoid mode score will generally raise more alerts, requiring more resources.
2. Look at the deployment environment. What base image are we using? Is the image supported by the scanner?
3. Based on risk (#1) and base image (#2), use the benchmark results (Table 3) to select the appropriate scanner.
4. From our experience, no image had zero vulnerabilities. A lack of vulnerabilities points to configuration problems or an unsupported image.
5. Combining multiple scanners in a CI/CD pipeline is a good idea. In paranoid mode, we suggest using the union of all scan results. In relaxed mode, use the intersection.

If the evaluated scanner is not in the benchmark, it can be added. Section 6 has links to the open-sourced benchmark.

5 Related Work

Research on benchmarking security tools is limited. El et. al. [9] discuss benchmarks for web vulnerability scanners, a difficult task due to vast landscape of web applications as well as multitude of potential web vulnerabilities and vulnerability classes. Nevertheless, Chen [3] attempts to benchmark web vulnerability scanners. UBCIS appears to be the first work that benchmarks container image scanners, although the CIS benchmark does exist to address run-time container security configuration best practices [13].

Studies of vulnerability classification are common, especially ones examining vulnerability type [10, 19]. Moreover, meta-studies and surveys are available that analyze existing vulnerability classification schemes [16,25]. We are not aware of any studies that focus on the applicability of detected vulnerabilities. Our work focuses on the applicability, highlighting that mis-identification of vulnerabilities can be for several different reasons (see Section 2.1). Mapping of vulnerabilities to either `TPs` or `FPs` is an important environmental decision.

6 Conclusion and Future Work

In this paper we discuss UBCIS, a tool created to evaluate the precision, recall, and F-measure of image scanners against base image distributions. We used UBCIS, evaluating three scanning tools against three of the most popular base images. We created a judging process for candidate vulnerabilities, manually evaluated all identified vulnerabilities to determine their relevance. Evaluation results can be applied immediately.

Correlating vulnerabilities between libraries will give a more granular picture of scanner detection. Such correlation is future work, as is extending the benchmark to deal with language specific package repositories (e.g., NPM, PIP, Ruby Gems). The dynamic nature of packages in these repositories, along with the number of vulnerabilities, presents a challenge.

Expanding the benchmark to include more images (e.g., CentOS) is future work. The open-source UBCIS benchmark will be available along with the vulnerability classification at `https://github.com/blackberry/UBCIS`, allowing others to use and build on UBCIS. Benchmark results will need to be regenerated as new vulnerabilities are judged, and as scanners improve. We have automated the process except for vulnerability judging.

References

[1] Alpine Security Databse v3.9 CVE List. YAML Object (accessed 14 May, 2020). https://github.com/alpinelinux/alpine-secdb/blob/master/v3.9/main.yaml.

[2] GitHub - Anchore Engine. Web Page (accessed 14 May, 2020). https://github.com/anchore/anchore-engine.

[3] Shay Chen. WAVSEP 2017/2018- Evaluating DAST against PT/SDL Challenges. Technical report, 2018. http://sectooladdict.blogspot.com/2017/11/wavsep-2017-evaluating-dast-against.html.

[4] GitHub - Clair. Web Page (accessed 14 May, 2020). https://github.com/quay/clair.

[5] CVE-2018-12886. Web Page (accessed 17 May, 2020). https://nvd.nist.gov/vuln/detail/CVE-2018-12886.

[6] CVE-2019-9192 - Debian Bug Tracker. Web Page (accessed 14 May, 2020). https://security-tracker.debian.org/tracker/CVE-2019-9192.

[7] Debian Security Bug Tracker. https://security-tracker.debian.org/tracker/.

[8] Docker Hub. Web Page (accessed 14 May, 2020). https://hub.docker.com/.

[9] Malaka El, Emma McMahon, Sagar Samtani, Mark Patton, and Hsinchun Chen. Benchmarking vulnerability scanners: An experiment on scada devices and scientific instruments. In *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 83–88. IEEE, 2017.

[10] Sophie Engle, Sean Whalen, Damien Howard, and Matt Bishop. Tree approach to vulnerability classification. Technical Report CSE-2006-10, Department of Computer Science, University of California, Davis, 2015. http://nob.cs.ucdavis.edu/bishop/notes/2006-cse-10/2006-cse-10.pdf.

[11] Google container registry. Web Page (accessed 14 May, 2020). https://cloud.google.com/container-registry.

[12] Why does my security scanner show that an image has CVEs? Web Page (accessed 14 May, 2020). https://github.com/docker-library/faq#why-does-my-security-scanner-show-that-an-image-has-so-many-cves.

[13] P Goyal. CIS Docker Community Edition Benchmark. Technical report, Center for Internet Security. https://www.cisecurity.org/benchmark/docker/.

[14] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. Over 30% of official images in Docker hub contain high priority security vulnerabilities. Technical report, BanyanOps, 2015. https://blog.banyansecurity.io/blog/over-30-of-official-images-in-docker-hub-contain-high-priority-security-vulnerabilities.

[15] Oscar Henriksson and Michael Falk. Static vulnerability analysis of Docker images. Master's thesis, Blekinge Tekniska Högskola, 2017.

[16] Shuyuan Jin, Yong Wang, Xiang Cui, and Xiaochun Yun. A review of classification methods for network vulnerability. In *2009 IEEE International Conference on Systems, Man and Cybernetics*, pages 1171–1175. IEEE, 2009.

[17] Alpine Linux Packages - libc-utils meta package. Web Page (accessed 14 May, 2020). https://pkgs.alpinelinux.org/package/edge/main/x86_64/libc-utils.

[18] Dan Lorenc and Maya Kaczorowski. Exploring container security: Let Google do the patching with new managed base images. Technical report, Google Cloud, 2018. https://cloud.google.com/blog/products/containers-kubernetes/exploring-container-security-let-google-do-the-patching-with-new-managed-base-images.

[19] Robert A Martin. Common Weakness Enumeration. Presentation Slides (accessed 15 May, 2020), May 2017. http://sqgne.org/presentations/2006-07/Martin-May-2007.pdf.

[20] National Vulnerability Database (NVD). Web Page (accessed 14 May, 2020). https://nvd.nist.gov/.

[21] Quay container registry. Web Page (accessed 14 May, 2020). https://quay.io.

[22] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on Docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280, 2017.

[23] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container security: Issues, challenges,and the road ahead. *IEEE Access*, May 2019.

[24] Byungchul Tak, Hyekyung Kim, Sahil Suneja, Canturk Isci, and Prabhakar Kudva. Security analysis of container images using cloud analytics framework. In *International Conference on Web Services*, pages 116–133. Springer, 2018.

[25] Anshu Tripathi and Umesh Kumar Singh. Taxonomic analysis of classification schemes in vulnerability databases. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pages 686–691. IEEE, 2011.

[26] GitHub - Aqua Security Trivy. Web Page (accessed 14 May, 2020). https://github.com/aquasecurity/trivy.

[27] Ubuntu CVE Tracker. Web Page (accessed 14 May, 2020). https://people.canonical.com/~ubuntu-security/cve/.

[28] HashiCorp Vagrant. Web Page (accessed 19 May, 2020). https://www.vagrantup.com/.

[29] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 491–501. IEEE, 2019.