Securing the Containerized Environment Along the CI/CD Pipeline

Mark Thompson
National Security Institute
Virginia Tech
Arlington, VA, USA
markt22@vt.edu

Michael Alex Kyer
National Security Institute
Virginia Tech
Arlington, VA, USA
makyer19@vt.edu

2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC) | 979-8-3315-0769-5/25/$31.00 ©2025 IEEE | DOI: 10.1109/CCWC62904.2025.10903704

***Abstract*—Containers are now ubiquitous in modern day software development. With all the necessary components pre-packaged and available within the container, containers present new security challenges because of their dynamic and ephemeral nature as well as a larger attack surface. Given the limited security knowledge of many software developers, we propose an automated and modular framework along the CI/CD pipeline that supports building and pushing container images to mitigate the security risks as much as possible. We tested our pipeline using containers with varying forms of vulnerabilities and found that the framework was successful in as much as it was able to identify known vulnerabilities in various components along the pipeline. Although it's not possible to possible to find every vulnerability (i.e., zero days), the modular framework supports multiple tools for wider coverage with a pipeline scheduler for continuous checking on whether or not new vulnerabilities arise. Even with this framework, it remains the responsibility of developers to use a variety of the most updated versions of tools along with community driven or custom rule sets that expand the capabilities of certain tools.**

Keywords—container security, CI/CD security, automated security, containerized applications

I. Introduction and Motivation

The adoption of containers has gained in popularity for application deployment due to their portable, flexible, and lightweight nature, allowing for faster deployment, integration, and scalability especially across cloud platforms. While containers have streamlined the software development and deployment process, they have also grown the potential attack surface to include the containers themselves and any additional infrastructure. Thus, software developers are now saddled with the additional responsibilities of not only ensuring the functionality, security, and integrity of their own applications, but also that of the containers and everything along the continuous integration and continuous deployment (CI/CD) pipeline that had usually been reserved for operations and infrastructure teams as well as cybersecurity experts in the past. This means that all vulnerability management, including patches and configuration settings, will now need to be managed by the software developer when building new image versions.

Since not all developers are fully aware of all the security risks surrounding their application and its containerized environment, we would like to provide an automated and modular framework that supports building a secure environment to minimize this risk as much as possible. Our initial research focus was to look at a practical framework that would ensure applications and tools within a container image approached a "zero vulnerability" state using as much open source and GitLab [1] as possible. While there are many aspects of containers with regards to their images, registries, orchestration, access management, and so forth, in this paper we focus on the container images themselves, specifically the containerized environment that allows us to build a secure environment for the deployment and running of applications.

Our research contributions lie within the modularity and scalability of the proposed framework. This research does not focus on quantitative comparisons of specific tools as it has been shown that vulnerability detection tools have different strengths and weaknesses [7, 9]. Instead, developers can use this framework and the "plug and play" nature of the types of tools mentioned to scale each module to improve the overall issue and vulnerability detection rate. Additionally, the positions of each module are flexible and only constrained by the build time of the container image. This allows the proposed modules to be placed into existing, complex CI/CD pipelines in an unobtrusive manner.

II. Related Works

The recent uptick in the use of containerized applications across countless organizations has put container security in the forefront of research topics of interest. Sachin Vighe made a strong case for the implementation of comprehensive security measures within the CI/CD pipeline to ensure the security posture of modern applications [2]. He identified key components in four major areas along the pipeline where security testing is performed and how important it is to identify essential security tools to mitigate any security vulnerabilities as early as possible in the development

979-8-3315-0769-5/25/$31.00 ©2025 IEEE

stages. Similarly, Bhardwaj et al. [28] proposed a security methodology along with CI/CD pipeline but used a "shift-left" approach to address runtime threats and image vulnerabilities early in the software development process. Our approach focuses specifically on the container image itself from the source to the build process but also adds an implementation framework that is both modular and scalable, allowing for a wide variety of tools to be used at each step.

Several authors have looked at the various areas in containerized applications that are subject to vulnerabilities, such as image vulnerabilities, runtime threats, network attacks, configuration errors, and access control issues, and then identified some of the best practices at a high level for each of these areas [3, 29]. While we agree with many of these best practices, our paper focuses solely on image security and provides the framework for implementing a number of these specific recommendations, plus several other techniques not included in these other papers.

In their discussions about integrating security into the CI/CD pipeline through DevSecOps, several authors endorse implementing security measures in all stages of the pipeline [4, 30, 31]. Abiola and Olufemi suggested the use of tools such as SAST that we have identified here but also recommend other tools like dynamic application security testing (DAST) and runtime application self-protection (RASP) as part of their automation strategy to address issues across different areas in the pipeline. Abiona et al. focused on the security practices of DevSecOps as an integral aspect in the software development process, rather than as a separate entity, including suggestions for industry-recognized tools and frameworks. Again, our focus is more directed towards container images themselves and how to effectively implement these practices within the pipeline.

Campo et al. [5] look at container security from a standards perspective, relying on NIST's Application Container Security Guide as the foundation for their work. Although they fell short of their goal of creating a blueprint for creating secure images using automated pipelines, they were able to provide recommendations to improve security processes when using containers. We also recognize that our loft goal of a zero-vulnerability state for container images may not be attainable, so we have developed our framework to be flexible and modular, allowing for multiple different tools to be executed at each stage, each potentially providing a diverse set of results, which when combined can provide additional confidence in the security of the container images.

III. Methodology

In our research to analyze the effectiveness of containerized images in preventing the introduction and exploitation of CVEs within containerized applications, we have developed a pipeline framework containing a series of flexible modules with the goal of promoting and ensuring security of containerized applications as shown in Figure 1.

We will run the first two modules after the commit operation, before the build process. We first focus on the application itself, analyzing the source code for errors, vulnerabilities, and standardization issues to improve code quality. A linter is used to detect any bad practices within the instructions on how to containerize the application [6]. Next, we use a static application security testing (SAST) tool to detect vulnerabilities found in the source code being stored within the container [7]. After the build process, a software bill of materials (SBOM) is generated for the users to maintain a record of each dependency and associated version loaded into the container [8]. A vulnerability management tool is then run on the built container to detect any known common weakness enumerations (CWEs) or common vulnerabilities and exposures (CVEs) [9]. Finally, a pipeline scheduler is implemented at regular intervals to automatically patch containerized applications with updates that may be related to security. It is important to note that the specific tools and services selected were largely chosen based on popularity and accessibility. Actual implementation of this framework could lead to pipelines that are highly variable and unique. Each

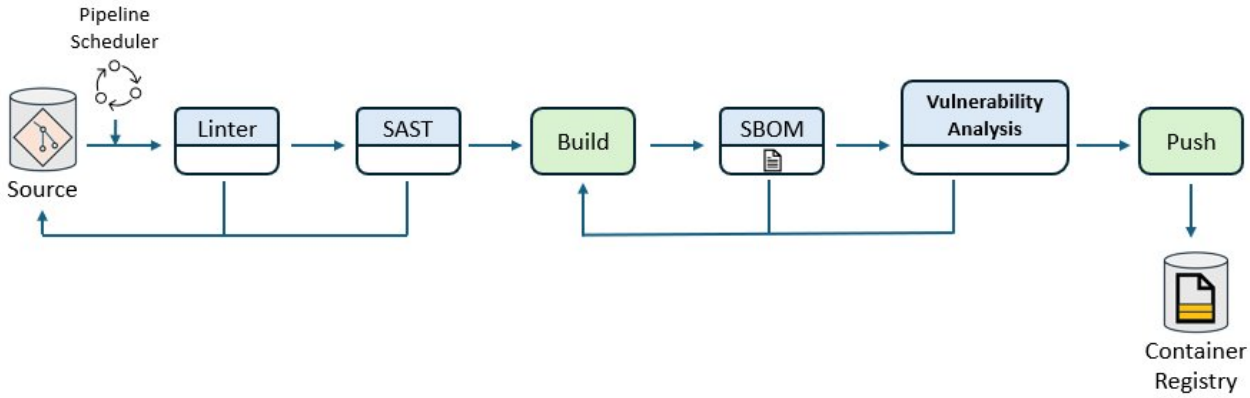

Figure 1. Pipeline Framework of Flexible Modules to Ensure Security of Containerized Applications.

module is flexible and scalable due to the plug and play nature of the related tools.

For the purposes of our research, Docker was used as the primary means of building, running, and storing container images [10]. Docker's popularity as the containerization tool with the highest market share made it a good option for our research [11]. Docker was also chosen as it is not restrictive to one operating system, it is free to use, it has set the standard for all other containerization tools, and it is the foundation for other popular options for the creation and management of containers.

Our framework was built and tested utilizing GitLab [1]. This decision was made in part to GitLab's prevalence in industry as a free version control system (VCS) that includes a collection of other tools, such as a variety of SAST analyzers, that can easily be incorporated into this framework. Additionally, the scalability of GitLab runners also allows for jobs of the same type, or tools within the same module, to be run in parallel. Each of the tools discussed in this module, including GitLab, is not essential to this flexible framework. Within each section, a series of options will also be presented that may be swapped in place of those used within this research. In the case of GitLab, it provides the VCS and CI/CD functionality. Alternatively, options such as GitHub [12] and BitBucket [13] can be used for the VCS functionality, and options such as Jenkins [14] and CircleCI [15] can be used for the CI/CD functionality.

A. Dockerfile Linter

The first step in the pipeline is to lint the Dockerfile. A Dockerfile provides the instructions for building a container. By linting this file, the instructions are scanned for programmatic errors that include security flaws. Several open-source options are available for Dockerfile linters, the most popular being Hadolint [16]. This framework was built with Hadolint due to its popularity, but other options include FROM:latest [17] and DockerfileLint [18]. Hadolint can be run from the command line, its own container, or from within the Dockerfile itself. A Docker-in-Docker approach was used to integrate Hadolint into this pipeline by spawning its own child container to run on the Dockerfile.

B. Static Application Security Testing (SAST)

SAST tools are capable of identifying a variety of vulnerabilities in the application, including buffer overflows and SQL injections. After the Dockerfile has been linted and the instructions secured, the next step is to analyze the application being placed into the container. To do this analyzation, SAST tools will be implemented as the next step of the pipeline to mitigate any issues related to the security of the application. After implementing a SAST tool, the Dockerfile and source code, the components that the developer is responsible for writing, will have been secured. There are multiple methods for implementing a SAST tool into a pipeline, and there are multiple SAST tools that can be implemented. Individual options of the latter will not be discussed here, as many options are exclusive to scanning source code written in a select few programming languages. However, the different types of SAST tools will be discussed.

1) Types of SAST Implementations

a) Open Source

Open source SAST tools are the focus within this research for a couple reasons. First, these tools are free, so there is no restriction on their use. Additionally, the source code is available online which often coincides with the ability to run through the command line which ensures they are able to be included within the pipeline. Inclusion can either be done manually, or with an existing or custom container including the tool and all required dependencies.

b) Repository Connectivity

Another approach that SAST tools utilize involves the connection to an external repository. The tool will be running externally from the pipeline, and it will contain project connections that update once the connected repository is pushed to. This type of SAST tool will not be explored in this research, but it is useful for separating the role of SAST from the pipeline to another service.

c) IDE Extension

SAST is also useful to be implemented during the development cycle of source code. Many SAST tools have an associated integrated development environment (IDE) extension that provides another way for developers to scan their code during development within the IDE. While this type of SAST tool can be useful, it is more difficult to connect to a pipeline approach to SAST.

2) Methods of SAST Implementations with a Pipeline

a) Pipeline SAST

It has become commonplace for CI/CD tools to provide their own extensions for SAST. Some examples of these tools include CircleCI and GitLab, the latter of which will be discussed more thoroughly. Within GitLab, additional jobs can be included from external configuration YML files found in another GitLab repo or on the local filesystem. GitLab provides a collection of these extensions that can be accessed publicly. One of these extensions includes the GitLab SAST, a combination of the Semgrep and FindSecurityBugs open source SAST tools [19]. By including this extension in one's repository, the project's directory structure will be searched recursively, and applicable files will be scanned. The results of the SAST scan will be output to the job's logfile.

b) Manual Commands

SAST tools can also be run manually within a pipeline. As most pipelines are run within a container, a SAST tool can be installed and configured with commands such as apt, pip, or yum. The steps for doing this are the same as the series of commands as found within the command line interface (CLI) reference written by the providers of the SAST tool or service.

c) Child Container

Many SAST tools have an associated container, typically Docker, that can be run in place of the CLI option. This method only requires the container service as the dependency, so nothing else needs to be installed into the pipeline. Utilizing a Docker-in-Docker approach, the pipeline container spawns a child container that automatically runs the SAST tool on the source code that is passed into it.

There are not only many different types of SAST tools, but there is a large variance in their capabilities as well [7]. It is important to consider the inclusion of a suite of SAST tools, as well as including a collection of custom rulesets to more broadly encompass the list of vulnerabilities able to be detected.

C. Software Bill of Material (SBOM) Generation

An SBOM is a report that lists all dependencies and versions within a given software component for libraries, tools, and processes used to develop a software artifact. In the case of a container, an SBOM is a list of all libraries, and their associated versions installed into the container after it is built. One of the goals surrounding an SBOM is to know the exact contents of a container. This way, if any security flaws are present, or become present later, the library with the issue can be upgraded or downgraded from its current version or removed entirely. Storage of these SBOMs also allows for older container versions, possibly still in use externally, to continue being monitored without pulling the entire image.

There are plenty of options available that offer SBOM generation capabilities, and there are plenty of formats available for storing SBOMs. Some open-source options include Syft [20], CycloneDx Generator [21], and SPDX SBOM Generator [22] to name a few. This research will implement Syft, an open-source SBOM generation tool developed by Anchore, due to its availability, ease of use, and popularity. Additionally, Syft outputs can be configured for either CycloneDx or SPDX in addition to its own TXT or JSON formatting.

D. Vulnerability Management Tool

A vulnerability management tool is used to scan applications for security flaws. In this case, a tool is required that can scan containerized applications. This step is directly related to the SBOM generation as each library within the container will be scanned, and if the version of said library contains a vulnerability, it will be reported.

Like the tools for SBOM generation, there are many options available. Grype [23], Trivy [24], Dagda [25], and Clair [26] are just a handful of options. The focus of this research will be on Grype, another product developed by Anchore that is often paired with Syft. Grype can scan containers on its own, but it can also scan the SBOMs produced by Syft allowing for legacy applications to be scanned as well.

It is important to note that, like SAST tool selection, one vulnerability management tool might not detect every vulnerability. Oftentimes, discrepancies among different vulnerability reporting tools occur, so it is usually recommended to incorporate more than one tool in this part of the framework. Scanning containerized applications against a series of vulnerability management tools potentially increases the number of vulnerabilities that can be detected, but it does come at the cost of duplicate results and more false positives.

E. Pipeline Scheduler

The final step of our pipeline framework is the schedule. This component allows containers to be rebuilt at regular intervals. Common intervals include daily or weekly depending on the size of the container and doing so allows for the inclusion of updated base containers as well as rescanning pre-existing components for potentially new vulnerabilities. The scheduler ensures this framework to be cyclic as opposed to linear which continuously ensures the security of containerized applications.

IV. Results from Testing the Pipeline Scheduler

The pipeline framework was implemented alongside a simple pipeline to build and push a Docker container to a registry. Some pipeline steps will occur prior to the container being built, while others will occur after the container is built but before it is pushed to the container registry. The repository being tested includes a series of Dockerfiles with known warnings pertaining to best practices. Of these Dockerfiles, one pulls from a secure image, “ubuntu:oracular”, and two from an insecure image, “ubuntu:kinetic”. The second insecure image is modified by the Dockerfile with the goal of making it more secure. Additionally, a Python script is contained within the repository that includes a known vulnerability.

The pipeline begins by linting the Dockerfile(s). A child Docker container is spawned which runs the external Hadolint program on the Dockerfile to be scanned. Any linter violations detected by this tool are reported to the job log or applicable console. Figure 2 is an example linter report of the improved Dockerfile that was pulled from an image known to be insecure.

```
$ docker run --rm -i hadolint/hadolint < ./insecure/ImprovedDockerfile
Unable to find image 'hadolint/hadolint:latest' locally
latest: Pulling from hadolint/hadolint
db4123164570: Pulling fs layer
db4123164570: Download complete
db4123164570: Pull complete
Digest: sha256:…
Status: Downloaded newer image for hadolint/hadolint:latest
-:3 DL3002 warning: Last USER should not be root
-:5 DL4006 warning: Set the SHELL option -o pipefail before RUN with a pipe in it. If you are using /bin/sh in an alpine image or if your shell is symlinked to busybox then consider explicitly setting your SHELL to /bin/ash, or disable this check
-:5 SC2010 warning: Don't use ls | grep. Use a glob or a for loop with a condition to allow non-alphanumeric filenames.
```

Figure 2. Linter Report from Image Known to be Insecure.


```
"version": "15.1.4",
"vulnerabilities": [{
"id":…,
"category": "sast",
"name": "Improper neutralization of special elements used in an OS Command ('OS Command Injection')",
"description": …,
"cve": "semgrep_id:python_exec_rule-subprocess-call-array:11:11",
"severity": "High“
},{
"id":…,
"category": "sast",
"name": "Improper restriction of XML external entity reference",
"description": …,
"cve": "semgrep_id:bandit.B314:8:8",
"severity": "Medium",
},{
"id": …,
"category": "sast",
"name": "Improper certificate validation",
"description":…,
"cve": "semgrep_id:bandit.B323:6:6",
"severity": "Medium"
}}
```


Figure 3. Results of SAST Scan of Repository's Source Code

The next step of the pipeline, also prior to building the container, is static analysis on the repository's source code. Tests were run utilizing both built-in pipeline SAST tools and manually installed SAST tools. The SAST tool used in both cases was Semgrep as it is an open-source option that works on Python files [27]. The built-in SAST reports vulnerabilities in a JSON artifact, while the externally installed SAST is reported within the job's log. The results of the scan can be seen in Figure 3.

The SAST tool and linter are utilized prior to building the container as it allows for early reporting and error reporting. Depending on the chosen process, the pipeline can be failed at any job after it is run. Therefore, it is possible to prevent the container from being built if there are any violations detected during the static analysis or linting process. Additionally, these jobs can be configured to allow for failure or only fail with violations of a certain severity. This customization allows developers to be as strict or lenient when it comes to security practices of their containers. In general, a stricter approach to security by failing the pipeline for each detected violation ensures greater security within the containers overall.

After the static analysis and linting jobs are successful, the pipeline is able to build the container. This step creates the image but does not deliver it to a container registry. In between those steps, an SBOM is generated, and a vulnerability analysis is run on the newly built container. This way, the container's delivery can be prevented if any vulnerabilities are found. These steps occur within the same job, so the container is only built once prior to being pushed.

```
$ syft zerocve:improvedinsecure
NAME VERSION TYPE
adduser 3.121ubuntu1 deb
apt 2.5.3 deb
base-files 12.2ubuntu3 deb
base-passwd 3.6.0 deb
bash 5.2-1ubuntu2 deb
bsdutils 1:2.38-4ubuntu1 deb
dash 0.5.11+…ubuntu1 deb
debconf 1.5.79ubuntu1 deb
debianutils 5.7-0.3 deb
diffutils 1:3.8-1 deb
dpkg 1.21.9ubuntu1 deb
e2fsprogs 1.46.5-2ubuntu2 deb
findutils 4.9.0-3ubuntu1 deb
gcc-12-base 12.2.0-3ubuntu1 deb
…
```

Figure 4. SBOM Report Using Syft After Building Docker Container

The SBOM is generated immediately after building the Docker container. The report is stored within the console log but can also be sent to a file. Figure 4 shows

the example SBOM generated on the improved insecure container. Each of the libraries included within the container are reported alongside their version and type. For the stripped version of the base Ubuntu image, the types are exclusively debians (deb), but there can be others present such as python, jar, rpm, etc. Storage of the SBOM is important to keep track of dependencies within one's software, especially older, fielded software. Additionally, vulnerability analysis can be conducted on SBOMs to check if new vulnerabilities have arisen since the image was last scanned.

Once the SBOM is generated for the container image, a vulnerability analysis should be conducted. This step is critical in ensuring the security of one's container. CVEs present in one's container can be exploited by adversaries to gain access to the functionality or information within, and the severity of that risk is enhanced once the container is fielded. A vulnerability analysis was conducted for each of the containers. While Grype has several fail-on severity settings ranging from negligible to critical, we used medium as our threshold for error reporting. Figure 5 shows the output for running Grype on a secure image. Figure 6 shows the output for an insecure image. Figure 7 shows the output for the improved image that was initially more insecure.

After the containers are scanned for vulnerability, the developers can determine what severity of CVE causes the pipeline to fail. If the pipeline has not failed through this step, the container is then pushed to a registry so that it can be pulled down by others.

```
$ grype zerocve:secure --fail-on medium
No vulnerabilities found
```

Figure 5. Results of Vulnerability Analysis Using Grype on a Secure Image

```
$ grype zerocve:insecure --fail-on medium
NAME VERSION TYPE VULNERABILITY SEVERITY
coreutils 8.32-4.1ubuntu1 deb CVE-2016-2781 Low
gpgv 2.2.35-3ubuntu1 deb CVE-2022-3219 Low
libc-bin 2.36-0ubuntu4 deb CVE-2016-20013 Negligible
libc6 2.36-0ubuntu4 deb CVE-2016-20013 Negligible
…
login 1:4.11.1+dfsg1-2ubuntu1.1 deb CVE-2023-29383 Low
passwd 1:4.11.1+dfsg1-2ubuntu1.1 deb CVE-2023-29383 Low
discovered vulnerabilities at or above the severity threshold
```

Figure 6. Results of Vulnerability Analysis Using Grype on Insecure Image

```
$ grype zerocve:improvedinsecure --fail-on medium
NAME VERSION TYPE VULNERABILITY SEVERITY
libc-bin 2.36-0ubuntu4 deb CVE-2016-20013 Negligible
libc6 2.36-0ubuntu4 deb CVE-2016-20013 Negligible
login 1:4.11.1+dfsg1-2ubuntu1.1 deb CVE-2023-29383 Low
passwd 1:4.11.1+dfsg1-2ubuntu1.1 deb CVE-2023-29383 Low
```

Figure 7. Results of Vulnerability Analysis Using Grype on Image Initially More Insecure

Externally a pipeline scheduler is implemented to repeatedly run this pipeline at a given interval to ensure they are up to date. One month after the initial test, the pipeline was rerun to simulate a scheduler component. Figure 8 displays the output of running Grype on the updated "secure" container. One can see that a multitude of CVEs were found, where once there were no vulnerabilities reported.

```
$ grype zerocve:secure --fail-on medium
NAME INSTALLED FIXED-IN TYPE VULNERABILITY SEVERITY
coreutils 9.4-3.1ubuntu1 deb CVE-2016-2781 Low
gpgv 2.4.4-2ubuntu18 deb CVE-2022-3219 Low
libc-bin 2.40-1ubuntu3 deb CVE-2024-33602 Medium
libc-bin 2.40-1ubuntu3 deb CVE-2024-33601 Medium
libc-bin 2.40-1ubuntu3 deb CVE-2024-33600 Medium
libc-bin 2.40-1ubuntu3 deb CVE-2024-33599 Medium
libc-bin 2.40-1ubuntu3 deb CVE-2016-20013 Negligible
libc6 2.40-1ubuntu3 deb CVE-2024-33602 Medium
libc6 2.40-1ubuntu3 deb CVE-2024-33601 Medium
libc6 2.40-1ubuntu3 deb CVE-2024-33600 Medium
libc6 2.40-1ubuntu3 deb CVE-2024-33599 Medium
libc6 2.40-1ubuntu3 deb CVE-2016-20013 Negligible
libgcrypt20 1.11.0-6ubuntu1 deb CVE-2024-2236 Medium
libssl3t64 3.3.1-2ubuntu2 deb CVE-2024-41996 Low
discovered vulnerabilities at or above the severity threshold
```

Figure 8. Results of Vulnerability Analysis Using Grype with Pipeline Scheduler One Month After Initial Test

V. Conclusion

The recommended pipeline framework promotes security in all aspects of building a containerized application. Prior to the container being built, the instructions for building are evaluated with a linter, and static analysis ensures the source code being placed within the container. After the container is built, an SBOM is generated to keep a record of the container's contents, and the image is scanned for vulnerabilities present within its components. To maintain the security of the built container, this process is repeated at a given

interval with a pipeline scheduler to identify newly found vulnerabilities as well as retrieving updates to mitigate past and present vulnerabilities. Testing this pipeline alongside a simple Dockerfile and vulnerability riddled source code, three different, unrelated sets of violations were found. This reveals the importance of each stage of the pipeline, and how each step works together to secure each part of the containerization process.

Flexibility and scalability of the modules composing the pipeline have been shown to be crucial in promoting container security. Flexible modules ensure they can be comfortably placed within existing, complex CI/CD structures so long as they are respective of the position of the container build. Scalable modules allow for the inclusion of multiple tools of the same intent but different capability to achieve maximal detection and minimal presence of vulnerabilities within.

REFERENCES

[1] GitLab, "GitLab," gitlab.com. https://about.gitlab.com/ (accessed December 17, 2024).

[2] Sachin Vighe, "Security for Continous Integration and Continuous Deployment Pipeline," *International Research Journal of Modernization in Engineering Technology and Science*, March 2024, doi: 10.56726/IRJMETS50676.

[3] Yash Jani, "Security Best Practices for Containerized Applications," *Journal of Scientific and Engineering Research*, 2021, 8(8):217-221, August 2021, doi: 10.13140/RG.2.2.26095.04000.

[4] Olumide Bashiru and Olusola Gbenga Olufemi, "An Enhanced CICD Pipeline: A DevSecOps Approach," *International Journal of Computer Applications* (0975 – 8887), vol. 184, no. 48, February 2023, doi: 10.5120/ijca2023922594.

[5] Frank Campo, Howe Wang, and Joel Coffman, "Exploring Solutions for Container Image Security," 14th Annual Ubiquitous Computing, Electronics, & Mobile Communication Conference (UEMCON), October 12-14, 2023, doi: 10.1109/UEMCON59035.2023.10316032.

[6] Giovanni Rosa, Federico Zappone, Simone Scalabrino, and Rocco Oliveto, "Fixing Dockerfile smells: an empirical study," *Empir Software Eng* 29, 108 (2024), doi: 10.1007/s10664-024-10471-7.

[7] Gareth Bennett, Tracy Hall, Emily Winter, and Steve Counsell, "Semgrep*: Improving the Limited Performance of Static Application Security Testing (SAST) Tools," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE '24)*, June 2024, doi: 10.1145/3661167.3661262.

[8] Nobutaka Kawaguchi, Charles Hart, and Hiroki Uchiyama, "Understanding the Effectiveness of SBOM Generation Tools for Manually Installed Packages in Docker Containers," *Journal of Internet Services and Information Security (JISIS)*, vol. 14, no. 3, August 2024, doi: 10.58346/JISIS.2024.I3.011.

[9] Omar Javed and Salman Toor, "Understanding the Quality of Container Security Vulnerability Detection Tools," arXiv:2101.03844v1, January 2021.

[10] Docker, "Docker," docker.com. https://www.docker.com/ (accessed December 17, 2024).

[11] IBM, "What is Docker?" ibm.com. https://www.ibm.com/topics/docker# (accessed December 3, 2024).

[12] "GitHub," github.com. https://github.com/ (accessed December 17, 2024).

[13] Atlassian, "Bitbucket," bitbucket.org. https://bitbucket.org/ (accessed December 17, 2024).

[14] "Jenkins," jenkins.io. https://www.jenkins.io/ (accessed December 17, 2024).

[15] Circle Internet Services, "CircleCI," circleci.com. https://circleci.com/ (accessed December 17, 2024).

[16] "Hadolint," github.com. https://github.com/hadolint/hadolint/releases (accessed December 17, 2024).

[17] ReplicatedHQ, "FROM:latest," github.com https://github.com/replicatedhq/fromlatest.io (accessed December 17, 2024).

[18] ReplicatedHQ, "Dockerfilelint," github.com. https://github.com/replicatedhq/dockerfilelint (accessed December 17, 2024).

[19] GitLab, "Static Application Security Testing (SAST)," docs.gitlab.com. https://docs.gitlab.com/ee/user/application-security/sast/ (accessed December 17, 2024).

[20] Anchore, "Syft," https://github.com/anchore/syft (accessed December 17, 2024).

[21] "CycloneDX Generator (cdxgen)," github.com. https://github.com/CycloneDX/cdxgen (accessed December 17, 2024).

[22] Microsoft, "SBOM Tool." github.com. https://github.com/microsoft/sbom-tool (accessed December 17, 2024).

[23] Anchore, "Grype," github.com. https://github.com/anchore/grype (accessed December 17, 2024).

[24] AquaSecurity, "Trivy," github.com. https://github.com/aquasecurity/trivy (accessed December 17, 2024).

[25] E. Grande, "Dagda." github.com. https://github.com/eliasgranderubio/dagda (accessed December 17, 2024).

[26] QUAY, "Clair," github.com. https://github.com/quay/clair (accessed December 17, 2024).

[27] Semgrep, "Semgrep," semgrep.dev. https://semgrep.dev/ (accessed December 17, 2024).

[28] Bhardwaj, A. K., Dutta, P., and Chintale, P. (2024). Securing Container Images through Automated Vulnerability Detection in Shift-Left CI/CD Pipelines. *Babylonian Journal of Networking, 2024*, 162-170. https://doi.org/10.58496/BJN/2024/016.

[29] Michael Sanya Oluyede, Joseph Mart and Akinbusola Olusola et al. "Container Security in Cloud Environments," *ScienceOpen Preprints.* 2024. doi: 10.14293/PR2199.000730.v1.

[30] Olutwatosin Oluwatimileyin Abiona, Olutwatayo Jacob Oladapo, Oluwole Temidayo Modupe et al. "The emergence and importance of DevSecOps: Integrating and reviewing security practices within the DevOps pipeline," in *World Journal of Advanced Engineering Technology and Sciences*, 2024, 11(02), 127-133. doi: 10.30574/wjaets.

[31] Dinesh Reddy Chittibala, "DevSecOps: Integrating Security into the DevOps Pipeline," in *International of Science and Research (IJSR)*, Volume 12, Issue 12, December 2023. doi: 10.21275/SR24304171058.