Knowledge base driven pipelines for security enforcement

*Anđela Trajković[1[0000-0002-0223-1756]], Milan Stojkov[1[0000-0002-0602-0606]], Miloš Simić[1[0000-0001-8646-1569]], and Goran Sladić[2[0000-0002-0691-7392]]

The University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia
{trajkovic.andjela, stojkovm, milos.simic, sladicg}@uns.ac.rs

Abstract. A lack of security controls in the system may be a potential point of attack to exploit the system's vulnerability. Unfortunately, security controls are added as an afterthought when all functionalities are implemented, which leads to difficulties adapting to the software's rigid policies, decreased performance, and increased costs. Furthermore, even after adjusting those policies, using an application that only partially fulfills some desired security requirements is difficult. The solution for decreasing time on adapting security mechanisms and minimizing weak points in the system becomes integrating security as a building block of development and maintenance, known as the DevSecOps concept. In this paper, we illustrate the importance of continuously providing protection in containers and reducing the risk of unwanted application attacks by integrating a secure pipeline in the earliest stage. The proposal is to automate the pipeline by combining security tools with database knowledge in a development process. The database knowledge will provide security policies that can be applied to a specific pipeline stage. This paper presents an approach to minimize vulnerabilities and code flaws by practicing DevSecOps, which also requires collaboration and communication between development, security, and operations teams, which increases the software's overall development efficiency.

Keywords: DevSecOps, security pipeline, vulnerability, container, security policies, database knowledge.

1 Introduction

In the business world, increased demand for automatization is caused by the need to make software development more customer-oriented. As a result, user

feedback becomes one of the principal criteria of software quality. Likewise, adaptation to a large audience has increased the number of users of the application. That requires more space for data stored and managed on a remote server. Remote servers are just remote physical machines whose hardware capacities can be distributed to multiple environments or users. The process of partitioning remote servers into virtual servers to efficiently organize hardware usage is called virtualization. Virtualization allows applications to be hosted on remote servers due to the lack of physical resources [1].

Two types of technologies provide server virtualization: hardware-level and operating system virtualization. Hardware-level virtualization involves a hypervisor that creates and virtualizes server resources across multiple virtual machines. Each piece of hardware, a virtual machine (VM), runs its operating system, own libraries, and dependencies. In contrast, operating system virtualization virtualizes resources only at the OS level. It encapsulates the standard process of the operating system and its dependencies to create containers managed by the host OS kernel [2]. In addition, they enable virtualization at low costs and better performance compared to virtual machines. Containers gained their popularity with the use of Docker [3].

Containers, with their efficiency, contribute to more easily transferring data and programs. As previously mentioned, user feedback is one of the main criteria for software quality. That requires continuous communication with the client. The best practices for frequent communication are continuous integration and delivery of software with agility, namely Development and Operations (DevOps) [4].

In opposition to VM, the container is efficient regarding data transfer [5]. However, the efficiency of securely deploying software with containers has declined significantly. Building security from the beginning of software development can reduce the space for security attacks. Augmenting the software with tools for scanning, monitoring security aspects, and updating features in the earliest stages makes it easier to maintain security in the application.

Considering the increased violation of sensitive data and ways of applying security in the industry, we want to define a minimum set of security requirements to improve protection of the whole system build cycle. This research describes the significance of continuously providing security in containers and reducing the risk of attacks by adding knowledge database with security policies [6] to the pipeline. Continually adding safety measures to software is automated by the DevSecOps pipeline [7].

In the pipeline, security tools and rules are used. Tools can automate testing and scan for vulnerabilities. Rules can enforce security policies and ensure that applications and systems are designed and deployed securely. Security policies can be enforced by blocking, notifying, or allowing specific actions based on the defined rules. Combining tools and rules in the software life cycle development makes adopting security easier and faster.

The research in this paper aims to answer two questions:

- Is it possible to make a support system for security-related activities in DevOps by relying on previous research and best practices?
- If so, can the security be applied using a rule-based system in the pipeline where domain knowledge is stored as formally defined rules?

The methodology consists of the following:

- Review of related work concerning the DevSecOps pipelines and expert systems designed to automate solving the security problems during the system development.
- Checking tool deficiencies to create and combine rules to complete the pipeline.
- Combining existing rules with existing deficiencies to complete the security requirement.

The following section provides an overview of related works with the research questions and methodology. The third section provides CI/CD pipeline and database knowledge. The security pipeline is described in the fourth chapter. The discussion is in the fifth chapter. The conclusion of this paperwork is presented in the last chapter.

2 Related work

This chapter presents a comparative analysis of the related and similar works. Bernardo and Giovanni [8] suggest a holistic approach and framework for the development and execution of trustworthy Infrastructure as Code (IaC), being secure, integral, and self-healed. The holistic approach mentions the challenges in the DevSecOps pipeline using database knowledge. It suggests using this pipeline in various areas of information technology.

Andel and Bill [9] describe the power of the DevSecOps pipeline by providing a way to optimize the application development process. In that publication, the authors focus on tools for each process, analyzing the pipeline and their advantages and disadvantages. The following processes are considered: Static Application Security Testing (SAST), Software Composition Analysis (SCA), Container Security, Dynamic Application Security Testing (DAST), and Interactive Application Security Testing (IAST).

Ramaj et al. [10] recommend automating the preparation of documentation using a DevSecOps pipeline so that documentation is created and updated whenever code or configuration changes are made. The process consists of using the tools to gather data from a system's settings and code and then updating the documentation with that data.

The authors discuss the difficulties in maintaining compliance in DevSecOps. They recommend using a systematic literature review to find the best ways to include compliance needs in a DevSecOps process. They identified several techniques that can assist organizations in maintaining compliance while implementing DevSecOps, including embedding compliance requirements into the design, automating compliance checks, utilizing compliance frameworks and standards, and adopting a risk-based approach [11].

Authors in [12] conducted a systematic literature review of 54 peer-reviewed studies. They apply the thematic analysis method to analyze the extracted data to find a balance between the speed of delivery and security, significant issues

practitioners face in the DevSecOps paradigm. Also, they mention some of the DevSecOps pipeline challenges.

Rahman et al. [13] consider the system architecture in three areas: continuous integration, continuous deployment, and continuous compliance with security requirements. Each of these areas encompasses a separate target for DevSecOps implementation. This study points out any gaps that would require further investigation.

Authors in [14] systematically explore experiences in utilizing security practices. They state the importance of introducing security in the earlier stages.

All these papers present some of the security requirements. A large part of the publications systematizes the shortcomings and vulnerabilities in the system. We used that kind of paper to gather knowledge in our knowledge base. Principally, they focus on one aspect of the security pipeline (authorization, authentication, etc.). In our work, we show how to connect the DevSecOps pipeline with rules (database knowledge), using all researched security weaknesses highlighted in the following chapters. After adding tools in DevSecOps, we add database knowledge for some features that tools do not support. We are aware that we cannot solve all security problems through a custom project, and therefore, in the end, we propose generating and shipping Software bills of material (SBOM) as a mandatory step. SBOM identifies and lists software components, information about those components, and supply chain relationships between them [15].

3 CI/CD pipeline and database knowledge

DevOps is a set of practices for automating software development processes. The most famous DevOps practices are CI/CD, Continuous Testing, Continuous Monitoring, Continuous Security, and Continuous Feedback. DevSecOps is an ideology for improving software development practice DevOps by integrating security, known as “shifting security on the left”.

The CI/CD pipeline is a part of the DevSevOps pipeline that actually represents the automation of the entire software development. The DevSevOps pipeline is composed of many sets of activities that a developer needs to perform to deliver a new version of a software product. Each set of activities is considered a stage (or phase) of the pipeline. Stages are planning, developing, building, testing, releasing, delivering, deploying, operating, monitoring, and feedback *Figure 1.*

Rule-based systems can supplement security best practices by following the standards and system gaps. Possible gaps in internal components are insufficient configuration and vulnerable code, whereas in external components are known vulnerabilities that have not been resolved, but the developer should know that they exist and that they can be avoided. More extensive security pipeline and the database knowledge is presented in the following chapter.

Earlier, it was mentioned that we would talk about virtualized applications, where we can use containers or virtual machines. Because of the prevalence of usage containers, we narrow the research to the security in the containers.

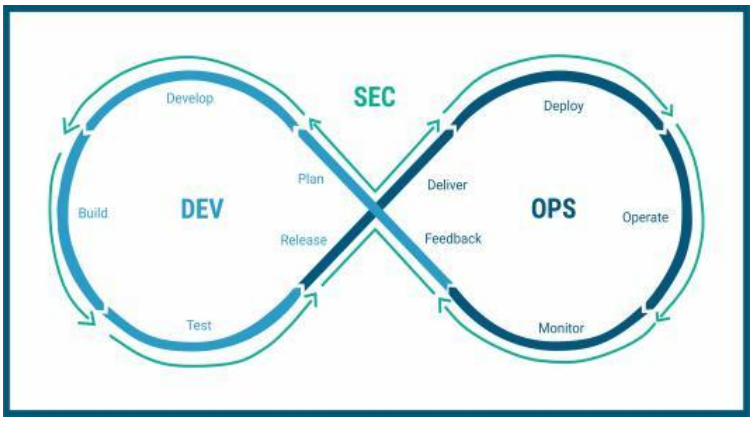

Figure 1 DevSecOps stages

The container represents the running image. Image is defined by the packages, source code, dependencies, and libraries it uses. It consists of a series of steps that the OS executes to run the application. The steps are interdependent and represent a step-by-step guide for building the software from the most basic to the most complex configuration. The containers are the focus of our proposed pipeline described in the next section.

4 Security Pipeline

Table 1 presents the overview of our proposed pipeline. The table has three columns: stage, security approach, and rule. The stage column represents all stages in the pipeline. The security approach represents a security strategy that can be applied at a certain stage, and the last column is a rule that can be triggered to solve or warn about issues in the system.

Rule-based systems can encompass the knowledge of the entire DevSecOps in one place. They enable the integration of the best security practices with tools and software. They are based on rules, which represent knowledge for detecting or solving security problems. Rules can be chained. We use forward chaining, where the rules are chained in the appropriate order, and respectively the output of one rule will represent the input of another rule. The database is located on a remote server. Used rule engine in the system is Drools [17]. The knowledge base with pipeline can be stacked in several ways, and one of the most commonly used is CI/CD pipeline.

Threats and solving security issues will be reasoned with a chained set of rules. The main chain represents the entire pipeline, which triggers an IMAGE VULNERABILITY side chain for reasoning.

Traditional DevSecOps involves integrating security practices into the development and deployment process, and all stages are listed in Table 1. The

Table 1 DevSecOps pipeline

Stage	Security approach	Rule
plan	Threat model	The beginning (PLAN phase) proposes to find bottlenecks using Threat modeling. Threat Model can envision the attack scenario and help to prevent most lacks and risks from the attack. The first rule triggers the tool for Threat modeling to identify the threats in the source code and environment. Output from this rule must be solved manually due to the poor compatibility of existing automation tools. As the Threat Model is the foundation of security by design, this can be tolerated for now. **Input**: Access to network and code (requirements). Check Network model and source code. **Output**: Warn admin about outputs.
sdevelop and build	Static analysis	This rule setup a quality gate for a number of vulnerabilities and code smells, above which, if the number of violations exceeds, it is not possible to proceed to the next phase. The focus is on the security rules, and it can be enriched with clean code rules, code style and conventions, bug detection, performance and efficiency, test coverage, documentation and comments, architecture and design, etc. Trigger on every week. Run SAST tool. **Output**: Warn about output; if it exceeds the upper threshold of the quality gate, then it cannot proceed to the next stage.
		Trigger on every week. Run SCA tool. **Output**: Code quality. Warning.
		IMAGE VULNERABILTY side flow start **Input**: Images Run a tool for updating image vulnerability to check if known vulnerabilities are solved. **Output**: If the vulnerability is resolved in the National Vulnerability Database (NVD)[16], the image will be recovered from the vulnerability.
		Trigger on every week. **Input**: Images Run Container scanner tool, to identify and address a vulnerability in the container. **Output**: Show all vulnerabilities. Run next rule.
		Input: Vulnerability image (output from the previous rule) 1. If vulnerability is in base image. **Output**: Warn about existing risks, for explicit libraries.

		Input: Vulnerability image (output from the previous rule) 1. If vulnerability is not in base image. **Output**: Warn about existing risks and recommend commands to DevOps team members to change or remove the layer(s). This step is up to the developer to consider the needs of the software. **IMAGE VULNERABILTY side flow end**
		Input: Configuration. Run script with commands for checking all the orchestrators. **Output**: If none of them are not triggered, trigger the rule for setup orchestrator, in order to setup and verify orchestration environment.
		Input: Configuration. Check if configured the least privileges, with script for checking statuses of Capabilities, Seccomp, SELinux, and AppArmor. **Output**: If they are not, this rule automatically setup profiles in the orchestrator: Capabilities, Seccomp, SELinux, and AppArmor [18].
		Input: Request. If DNS is on the blocklist. **Output**: Block request and user, then alarm admin.
		Input: Request. Run the tool for analyzing the packets (Wireshark) **Output**: Captured packets. Manual check for encrypted communication (HIP, TLS/SSL, SSH...). If not exist, warn as insecure communication.
	Security as Code	**Input**: Configuration. Run command for checking is image run in the privileged mode. **Output**: Warn if it is run in privileged mode.
		Input: Container. Check mode of the running container. **Output**: Warn if it is run in privileged mode.
test		Trigger on every week. Run Unit, Integration, and e2e tests automate. **Output**: Possible attacks.
		Trigger on every week. **Output**: Notify security admin to run Dynamic Application Security Testing tools, Interactive Application Security Testing tools and Penetration Testing to simulate attacker and check the application in runtime, to detect SQL injection, APIs, user authentication and authorization.
release	Digital Sign	**Input**: Image. Sign image. **Output**: Notify the image is signed image.
deliver	Secure Transfer	**Input**: data, code, files. Run Secure Transfer tool. **Output**: Secure transfer.

deploy	Security configuration and Scan	**Input**: Configuration. Check who can pull and push source code from the repository. In terms of managing, who can modify and review existing code. **Output**: If there is no limitation, warn the admin to ensure an appropriate access level.
operate	Security Patch	**Input**: data, code, files. Run Secure Patch tool. **Output**: Possible attacks.
monitor	Security Monitor	Run Logs tools **Output**: Logs. Admin can manually analyze all events in the system.
		Input: / Run Metrics tools. **Output**: Metrics. Admin can manually analyze performance in the system.
feedback	Security Analysis	Future work.

following is a comparison of a typical flow, with the specificities of the flow we implemented, including their advantages and disadvantages.

The traditional PLAN phase is manual and includes implementation user and security requirements, Data Flow Diagrams, and Threat Modeling frameworks (vulnerability frameworks like OWASP TOP 10 vulnerabilities [19], MITRE ATT&CK [20], or STRIDE [21]). This phase is also manual in our pipeline implementation and requires a security expert. The specificity we introduce is expanding the boundaries of Threat Modeling (using vulnerability frameworks like OWASP TOP 10 vulnerabilities), including the attack vectors between hosts, kernel, container networks, virtual machines, and physical hardware.

DEVELOP and BUILD phase requires activities such as Static Application Security Testing (SAST), Static Composition Analysis (SCA), orchestration, checking least privileges, and encrypted communication. In our flow, we wanted to reduce vulnerabilities in the container's additional layers by allowing the possibility to remove unused vulnerable packages. Still, removing the exposure in the image is impossible because it can disrupt the system's operation (especially if it is in the base image). The base layer represents the foundation of the Docker image and contains the underlying operating system and any essential dependencies or packages required by the application. The additional layers customize the base image, and if those additional layers are not needed per system requirements, then the developer can decide to remove them.

In this phase, at the very beginning, our pipeline triggers SAST tools (SonarQube [22]) and SCA (snyk [23]). The SAST tool finds threats and wrong implementation in the code, whose outputs must be solved manually or connected to other tools. SCA tool scans for vulnerabilities, but we suggest the developer can mitigate those vulnerabilities depending on the system requirements with the IMAGE VULNERABILITY flow. After finishing the IMAGE VULNERABILITY set of rules, the main flow continues, and the pipeline triggers rules for the BUILD stage.

In the build stage, we automatically check if orchestration exists. Orchestration is important because it provides a higher level of isolation that will help the user to save sensitive data. After that, it enables checking and setting the least privileges by automatic setup (declarative in the configuration) profiles such as Capabilities, Seccomp, SELinux, and AppArmor [18]. Capabilities provide a way to divide privileges in the container. Seccomp profiles were introduced in Docker 1.10 to limit the system calls that can be made by the container. This feature adds another level of security as it ensures that the container can do only what it needs to do [18]. The other two provide access control mechanism in the entire environment.

Having root privileges when running a container makes accessing the host OS and other containers on the hosted OS easier. Accordingly, we want to ensure the container does not have root privileges to run by warning the security admin.

Encryption is one of the main steps of secure communication and can be conducted with a secure protocol. First, the pipeline triggers a rule for running the tool for analyzing the packets and checking manually where admin after that rule should answer is communication secure.

The test phase traditionally includes DAST and Penetration Testing, along with security smoke testing, security patching, and Interactive Application Security Testing (IAST). In this stage, we are focused only on automating e2e, integration, and unit tests. IAST, DAST and Penetration testing are configured to notify the security admin to test separately. Notifying an expert is currently the best option because it is challenging to automate this expertise.

The RELEASE phase uses Runtime Application Self-Protection (RASP) technologies to improve security. We are not adding RASP technologies, but we are signing the image to ensure authenticity, secure communication, and integrity of the image to enhance security, automatically with Notary tool [24].

The traditional DELIVER phase runs the Secure Transfer tool (rsync [25]), and so do we, to encrypt data in transfer. In the DEPLOY phase configuration should be checked and all flow should be scanned; we added checking who can push and pull from the safe repository (git), which increases the security level. Flow scan is planned in the future work.

OPERATE deploys software in the production environment and runs Secure Patch tools (Spacewalk [26]). We trigger the rule to run a secure patch tool to automate the patch management process. Adding tools for metrics and logs can generate a large data set, which security experts can analyze manually.

A FEEDBACK stage is out of scope and will be considered future work because it requires communication with clients, which is delayed until performance improves.

The emphasis is on the initial stages while analyzing all these outputs to automate self-healing would be in the further approaches. Tools used in pipelines generally ignore warnings. We automatically throw a warning for other output from the Threat model that we cannot solve with the rules, where the acceptance threshold would be configurable (e.g., 3 or 30 depending on the system, to have a supportive pipeline for security). When that threshold is crossed, we mark the pipeline as unstable so that the warning is not ignored. Also, there are alarms in the system that mark the pipeline as inconsistent. Each phase can be marked with an

error (where the pipeline is terminated), unstable (where the system gives a warning), or successful flag. If all pipeline stages are successfully finished, the pipeline is successfully executed, and the product can be used. A successful flow is executed up to the deploy phase and reports with SBOM. The stages deploy, operate, and monitor are optional and require communication with the client; these phases will be started depending on the client's requirements. This flow is applicable to the complete software solution delivered to clients. The solution applies to on-premises and enterprise applications. Cloud deployments are out of scope for the current pipeline.

With this pipeline, we establish a standard for a certain level of security in the system to force the user to accept it as the bare minimum security. The presented approach's advantages, disadvantages, and possible improvements are explained in the next section.

5 Discussion

The pipelines can be evaluated by different criteria, such as the execution time of individual phases or tasks, total execution time, or response to error resolution depending on the DevOps engineer and his expertise. The traditional pipeline and proposed pipeline enhanced with a knowledge-based system were evaluated by the total execution time. An internally developed micro-cloud system with 21 containerized services was used as part of the experiment. As expected, the execution time of the proposed flow was 22% slower but with better control over the individual tasks, especially for those that can generate warnings. The IMAGE VULNERABILITY side chain slows the pipeline by 9%, which we consider tolerable.

Automating the entire pipeline is hard because only some things can be solved automatically. It should also be borne in mind that the used tools have their shortcomings and that it is possible to attack any application that uses the services of a given tool for security implementation.

Currently, we do not examine whether this DevSecOps pipeline is complete. It is hard to complete because traditional DevSecOps predominately resides in the development stage. We can only write and use practices and methodologies to ponder as many security requirements as possible. Despite the obstacles, we strive to achieve this goal.

The major problem in the system is to protect sensitive data. By including the orchestration, we are adding one additional data isolation layer.

Threat Model tools identify external and internal threats, where the majority of checking criteria are made with best security practices, and we try to recognize their insufficiencies with rules. A disadvantage is that we cannot resolve all vulnerabilities, but we provide an option to remove the vulnerability if the system does not need the library (package) that contains the vulnerability.

We provide awareness of what should be scanned. Everywhere where a person can make a mistake is alarming. We still cannot fully remove the vulnerability.

Taking into account storing large amounts of knowledge in one place, the best usage of security policies is through a knowledge base. Knowledge stored in the form of the rule does not harm the development code. That allows clean code and separates it from cumbersome security checks.

We recommend using tools in different stages, but with some caution against impairing software performance. Using a large number of scanning tools and improving security aspects destroys performance, which is a trade-off.

Generally, the advantages of DevSecOps with database knowledge are:

- Increased productivity, speed, and agility between teams.
- Rapid changes.
- Detection and clarification of flaws in code at an earlier stage
- Early detection defects in code.
- Localization of a problem in code does not have to be hunted manually.

The use of a knowledge base helps a lot in covering security in one place and in developing security simultaneously with the project itself, but there needs to be a compromise.

The detected shortcomings also represent possible improvements. All warnings in our pipeline should automatically be repaired through rules (database knowledge). In the case of implementing a microservices architecture, there will be explicit communication between containers in the system, where the network bridge should be protected.

In this pipeline, we strive to follow the Zero Trust approach. Our system has verification, least privilege access, continuous monitoring (without automated analyzing), encryption, and data protection. In future work, we will investigate to add zero-trust network access checks and strive to micro-segmentation of the steps (for the purpose that every part has its control). Output from every stage should be analyzed automatically in purpose to resolve problems in the earliest stage and make a self-healing system.

6 Conclusion

Creating a pipeline by combining tools and rules as expert knowledge of security applications, we increase the degree of automation of application and development security on the left side. Of course, it should be borne in mind that the pipeline is not finished. DevSecOps is in the development stage and requires more time to automate each step, but we strive to achieve this goal.

The future work would be based on automating the rest of the pipeline. The final goal will be automatically analyzing and solving threats by using the rules in cooperation with machine learning techniques and working with security for multi-tenancy to enable the deployment of our pipeline to cloud-based applications.

References

1. Yussupov, V., Soldani, J., Breitenbücher, U., Brogi, A., & Leymann, F. (2021). From

Serverful to Serverless: A Spectrum of Patterns for Hosting Application Components. In *CLOSER* (pp. 268-279).
2. Sharma, P., Chaufournier, L., Shenoy, P., & Tay, Y. C. (2016, November). Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference* (pp. 1-13).
3. Docker Hub, https://hub.docker.com/, accessed January 2023.
4. Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. *Ieee Software*, *33*(3), 94-100.
5. Qadir, S., & Quadri, S. M. K. (2016). Information availability: An insight into the most important attribute of information security. *Journal of Information Security*, *7*(3), 185-194.
6. Jacob, R. J. K., & Froscher, J. N. (1990). A software engineering methodology for rule-based systems. *IEEE Transactions on Knowledge and Data Engineering*, *2*(2), 173-189.
7. Kudriavtseva, A., & Gadyatskaya, O. (2022). Secure Software Development Methodologies: A Multivocal Literature Review. *arXiv preprint arXiv:2211.16987*.
8. Alonso, J., Piliszek, R., & Cankar, M. (2022). Embracing IaC through the DevSecOps philosophy: Concepts, challenges, and a reference framework. *IEEE Software*, *40*(1), 56-62.
9. Bernardo, G. (2022). *DevSecOps pipelines improvement: new tools, false positive management, quality gates and rollback* (Doctoral dissertation, Politecnico di Torino).
10. Andel, B. (2022, October). Continuous Documentation: Automating Document Preparation with your DevSecOps Pipeline. In *2022 IEEE 29th Annual Software Technology Conference (STC)* (pp. 156-165). IEEE.
11. Ramaj, X., Sánchez-Gordón, M., Gkioulos, V., Chockalingam, S., & Colomo-Palacios, R. (2022). Holding on to Compliance While Adopting DevSecOps: An SLR. *Electronics*, *11*(22), 3707.
12. Rajapakse, R. N., Zahedi, M., Babar, M. A., & Shen, H. (2022). Challenges and solutions when adopting DevSecOps: A systematic review. *Information and software technology*, *141*, 106700.
13. Deshmukh, S. V., Ahire, D. S., Chavan, N. N., Bharambe, N. D., & Jain, A. R. Implementing DevSecOps pipeline for an enterprise organization.
14. Ur Rahman, A. A., & Williams, L. (2016, May). Software security in devops: Synthesizing practitioners' perceptions and practices. In *Proceedings of the international workshop on continuous software evolution and delivery* (pp. 70-76).
15. Muirí, É. Ó. (2019). Framing Software Component Transparency: Establishing a Common Software Bill of Material (SBOM).
16. NVD, https://nvd.nist.gov/, accessed January 2023.
17. Drools, https://docs.drools.org/5.4.0.Beta1/drools-expert-docs/html_single/, accessed January 2023.
18. Park, K., & Kim, B. (2020). Core Container Security Frameworks. *International Journal of Advanced Research in Engineering and Technology (IJARET)*, *11*(6).
19. OWASP Top Ten, https://owasp.org/www-project-top-ten/, accessed January 2023.
20. MITRE ATT&CK, https://attack.mitre.org/, accessed January 2023.
21. STRIDE,https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN , accessed January 2023.
22. SonarQube, https://www.sonarsource.com/products/sonarqube/ , accessed January 2023.
23. Snyk, https://snyk.io/ , accessed January 2023.
24. Notary, https://github.com/theupdateframework/notary, accessed January 2023.
25. Rsync, https://www.digitalocean.com/community/tutorials/how-to-use-rsync-to-sync-local-and-remote-directories, accessed January 2023.
26. Spacewalk, https://spacewalkproject.github.io/ , accessed January 2023.