Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines

Thorsten Rangnau
University of Groningen
t.rangnau@student.rug.nl

Remco v. Buijtenen
University of Groningen
r.m.van.buijtenen@student.rug.nl

Frank Fransen
TNO
frank.fransen@tno.nl

Fatih Turkmen
University of Groningen
0000-0002-6262-4869

***Abstract*—Continuous Integration (CI) and Continuous Delivery (CD) have become a well-known practice in DevOps to ensure fast delivery of new features. This is achieved by automatically testing and releasing new software versions, e.g. multiple times per day. However, classical security management techniques cannot keep up with this quick Software Development Life Cycle (SDLC). Nonetheless, guaranteeing high security quality of software systems has become increasingly important. The new trend of DevSecOps aims to integrate security techniques into existing DevOps practices. Especially, the automation of security testing is an important area of research in this trend. Although plenty of literature discusses security testing and CI/CD practices, only a few deal with both topics together. Additionally, most of the existing works cover only static code analysis and neglect dynamic testing methods. In this paper, we present an approach to integrate three automated dynamic testing techniques into a CI/CD pipeline and provide an empirical analysis of the introduced overhead. We then go on to identify unique research/technology challenges the DevSecOps communities will face and propose preliminary solutions to these challenges. Our findings will enable informed decisions when employing DevSecOps practices in agile enterprise applications engineering processes and enterprise security.**

***Index Terms*—DevSecOps, Dynamic Security Web Testing, Continuous Security, Continuous Integration**

I. Introduction

In the past decade, a great shift occurred in software development from creating *Software as a Product* (SaaP), that is executed as a single instance on customers' machines, towards providing *Software as a Service* (SaaS) where many users share instances that run on cloud infrastructure [1].

Such cloud services provide software practitioners with the ability to continuously improve their product quality by releasing frequent updates [2]. In order to manage these improvements efficiently, classical development (Dev) and operation (Ops) tasks were combined which resulted in a development concept termed DevOps [3], [2]. This concept is based on collaboration between the two former fields in all development stages and achieved by solving problems together, automating processes, and agree on mutual metrics to use when evaluating a system. DevOps defines four pillars that guide teamwork in modern software development: culture, automation, measurement and sharing (CAMS) [4], [5]. This agile development method enables software practitioners to test and deploy software versions in a much more frequent pace and hence respond to customers' demands rapidly. A prime example of this is Amazon, where a new version was released more than once per second [6].

While fast releases are considered to be beneficial to the quality of a product, they may also increase pressure on developers to finish their tasks more quickly. Studies such as Kraemer [7] revealed that tight schedules or high work load can lead to the accidental introduction of security vulnerabilities into software systems. Kraemer also states that the reason for the presence of vulnerabilities is a lack of security knowledge in DevOps teams. This affects the quality of security tests and hence diminishes the security of a system. In addition to this, cybercrime is increasing in recent years. For instance, the number of stolen or compromised records has been estimated to be increased by 133% from 2017 to 2018 [8]. Furthermore, security and privacy regulations such as the General Data Protection Regulation (GDPR) have been implemented in the EU in order to enforce security standards and punish companies harshly if these regulations are violated (e.g. [9]). All of these aspects show that the security concerns have become increasingly important.

This increased focus on security introduced a new field called DevSecOps, which attempts to integrate security (Sec) practices into DevOps [2]. Traditionally, security experts were organized into separate silos and security concerns were addressed after the actual design and development stages [8]. Similar to the inception of DevOps, DevSecOps attempts to promote collaboration between development, operations and security teams. DevSecOps establishes a proactive approach to limit the attack surface of the application [6] and entails considering security from the very beginning of the project [2]. However, the integration of security practices into modern software engineering creates several problems. Firstly, traditional security methods are not applicable because they cannot keep up with the agility and speed of DevOps. Secondly, very little is known about DevSecOps so far, as only a few studies were conducted on this topic [2]. Especially the lack of knowledge of when and where to use (existing) tools in automation is a considerable problem that prevents software practitioners from integrating security into their DevOps activities such as continuous integration and continuous deployment (CI/CD) [8].

2325-6362/20/$31.00 ©2020 IEEE
DOI 10.1109/EDOC49727.2020.00026

Until now, most research papers describe the principles, priorities and practices in DevSecOps. It appears that the automation principle is equally significant in DevSecOps as it is in DevOps. One key practice is continuously testing security. This enables security teams to keep up with DevOps and establishes fast, scalable and effective security tests [2]. However, most literature focuses on automatic security testing through static scans of source code (e.g. [10]). Although important, static tests cannot detect all security vulnerabilities in a system. In fact, static analysis is only able to find those vulnerabilities that can be derived directly from source code. These vulnerabilities are only a small subset of the ten most common vulnerabilities in web applications [11], such as *Components with Known Vulnerabilities*. Dynamic security testing on the other hand, where a system is attacked in a similar way as actual hackers would, is able to cover a much broader range of vulnerabilities. Literature such as [12], [13] describes how to execute these dynamic tests in a consistent, reproducible way. However, little is known about how to integrate this into the CI/CD pipelines commonly used in DevOps.

In this paper we conduct a case study where we apply three different testing techniques in CI/CD. This will enable us to identify pitfalls, challenges and shortcomings DevOps teams may encounter while automating security tests. The three dynamic application security testing techniques we integrated into a CI/CD pipeline are: Web Application Security Scanning (WAST) using Zed Attack Proxy (ZAP)[1], Security API Scanning (SAS) with JMeter [2] and Behaviour Driven Security Testing (BDST) using SeleniumBase automation framework.

The remainder of this paper is structured as follows: Section II provides an overview on automated security testing techniques and testing in CI/CD. The setup of the case studies is described in Section III, whereas Section IV depicts our results. In Section V we discuss our findings including a list of requirements for security testing in CI/CD and a description of challenges one encounters while addressing these requirements. We conclude in Section VII and provide an overview of future work.

II. Background

This section provides an overview on the background of continuous dynamic security testing. To this end, we will first address security testing techniques. Subsequently, we provide information on testing in CI/CD pipelines.

A. Security Testing Techniques

Most modern Web/Cloud applications can be tested for security flaws at the service, infrastructure, and platform levels [14]. We focus on the testing performed at the service layer. Dynamic application security testing (DAST) focuses on tests to determine how a running application responds to malicious requests. More specifically, attack scenarios are defined as test cases that consist of (crafted) requests and are sent to the system [13]. The challenge here is to send the correct (attack) requests and to identify the information within the response that indicates the presence of a vulnerability. DAST can be performed in a white-box setting where the application code is accessible or a black-box setting where the application code is unavailable. We assume the CI/CD pipeline is owned by the application owner and thus consider mostly the white-box case. In what follows, we summarize the dynamic testing methods considered in our study.

Inspired from [13], we consider three DAST techniques that can be automated: Web Application Security Testing (WAST), Security API Scanning (SAS), and Behaviour Driven Security Testing (BDST).

Web Application Security Testing (WAST): This testing technique is an automated web security test that attacks a web application through its user interface. It includes three steps performed by the WAST component: spider scan, active scan, and result reporting. The spider scan explores the whole application in order to determine all URLs/resources available. The active scan then performs malicious requests against every identified resource and evaluates every response of the application in order to determine possible security issues on the targeted URL. Once the active scan is completed, the results are aggregated into a report. Figure 1 illustrates the WAST technique. Usually, the scope of the scans and the attack scenarios can be configured. In addition, the security vulnerabilities can be categorized into different risk levels.

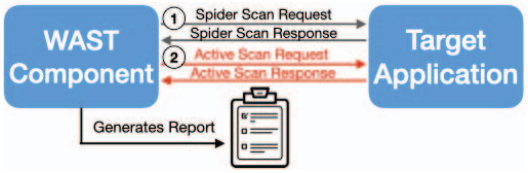

Fig. 1. Overview of WAST security testing technique: The spider scan determines all available components and the active scan attacks them.

Security API Scanning (SAS): The WAST technique scans the entire web application but may not detect flaws of the underlying back-end (web) services. Therefore it is highly recommended to test the web service through its APIs with SAS. This technique allows testing of every endpoint in great detail and can cover multiple security relevant cases such as authentication, input validation, or error handling. Figure 2 provides an overview of the SAS testing technique. In SAS, a parameterized request is generated and sent to the API of the web service that is under test through a *request component*. The input data can vary from credentials for authentication to malicious payloads such as SQL injection (SQLi). All the requests go through a *proxy component* that intercepts traffic between the request component and the target application. The proxy component evaluates all the intercepted traffic for any security issues. After the test is performed, the proxy component reports the result of the evaluation. SAS testing is especially useful when generated fuzz data is used as input. Fuzz data can be a list of the most common passwords, a bulk

[1]https://www.zaproxy.org/
[2]https://jmeter.apache.org

of random data in order trigger unexpected behavior of the system, or malicious input for SQLi.

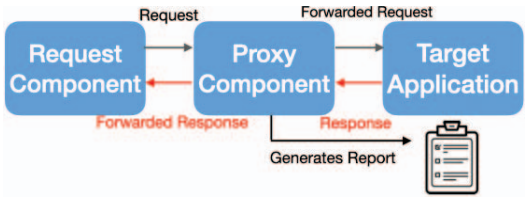

Fig. 2. Overview of SAS testing technique: Request component sends request through proxy component to target application. Proxy component evaluates traffic and generates report.

Behaviour Driven Security Testing (BDST): Behaviour Driven Development(BDD) is an extension of Test Driven Development (TDD) and follows the idea of integrating business insights into testing [15]. BDD uses a natural language approach in order to define behaviour and expected outcome of test cases. Behaviour Driven Security Testing (BDST) applies the idea of BDD to the domain of security testing for the added benefit that non-security experts can understand the security tests, further improving the collaboration between security experts and DevOps teams [12]. Additionally, BDST provides a dynamic security documentation of the whole software system due to the GWT (Given, When, Then) format of the test specifications. In UI testing, BDD frameworks are used to automate standard UI tests that mimic the user behavior [15]. This approach can be used to automate the execution of attack scenarios from the hacker's perspective. Because this technique is executed against the system as a whole, it enables the identification of vulnerabilities that target multiple entrypoints to the system. BDST combines several security testing techniques such as SAS or WAST in order to mimic attack scenarios by a hacker, as well as to find security issues during normal system usage [13]. An example of a BDST setup is shown in Figure 3.

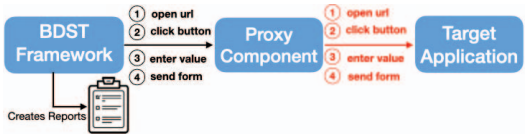

Fig. 3. Overview of BDST testing technique: The BDST framework sends behavior driven requests to the target application through the proxy component which then scans for security flaws.

B. CI/CD Pipelines

Continuous Integration (CI) and Continuous Delivery (CD) are software engineering processes used in DevOps in order to improve the efficiency of projects [16]. The CI/CD processes can be implemented at one of three overarching degrees of automation. The first covering development and testing (Continuous Integration), the second extending this with automated integration testing (Continuous Delivery) and the third adding continuous deployment to a production environment (Continuous Deployment). Figure 4 shows a CI/CD pipeline that has been extended with dynamic security testing. A more typical CI/CD pipeline only consists of stages 0 through 7 that are shown in the upper 2 rows of the figure. Later on in section IV-B a more detailed explanation is given on the addition of security testing.

The Continuous Integration stage starts with a commit, followed by a build of the modified application which is then verified using unit tests. When all test cases pass, the tested application is deployed to a testing environment. If Continuous Delivery is implemented, a set of automated acceptance tests is executed to verify that there are no regressions in the system's features. This step also helps to identify any errors that may occur due to a difference in run-time environment because the testing environment is usually a server with similar configuration to the production environment. Depending on the level of automation, this step can also involve manual testing and approval before the pipeline advances to the next stage. When all previous stages have passed, the system is automatically deployed to the production environment in the Continuous Deployment stage, where the users will have access to the new version. If a test fails, or when an error occurs during build or deployment, the pipeline is automatically stopped and developers are notified of the error. When a fix has been committed the pipeline will start all over again in order to test the entire application.

III. Case Study Methodology

This section aims to provide an overview of our goals and research question that we want to answer with the case study conducted in this paper. Next an outline of our approach is provided.

A. Goal and Research Question

The primary goal of this study is to identify the challenges and pitfalls of applying security testing of web applications and services in CI/CD pipelines. Moreover, the focus lies on dynamic testing techniques since this topic is only addressed in theoretical research. With this we provide guidance for development teams that are trying to shift more towards DevSecOps. In addition to this, we aim to shift research towards the practical challenges involved with continuous security.

RQ *How can we integrate DAST into CI/CD and ensure that DevOps requirements are met?*

With answering this research question we can shed light onto the hidden complexities of the practical workload required to achieve this integration. Through practical examples we illustrate the scale of these complexities, as well as preliminary solutions that aim to overcome them.

B. Approach

This section provides an overview of the approach used in our case study. In order to conduct this case study we (1) identify the requirements for successful integration of security testing tools, (2) describe the tools that were chosen for the integration and discuss their integration into CI/CD, (3)

investigate the performance of our implementation in a CI/CD pipeline and (4) provide an overview of the challenges that were encountered during implementation.

1) The requirements are determined using an iterative process. As a starting-point we use common requirements for DevOps. After creating an initial version of the CI/CD pipelines, the requirements were extended to reflect the changes needed to meet the DevSecOps goals. The final list of requirements is presented in section IV-A. Later on in section V, these requirements will be used to evaluate the implementation.
2) Three different methods for DAST were listed in section II. A number of tools are chosen in order to implement the three testing techniques in such a way that they satisfy our requirements. Later on in this section we provide a short description of each of the chosen tools. Furthermore, in Section IV more details are provided on how each tool is adapted in such a way that they can be executed inside a CI/CD pipeline.
3) With the implementation of our CI/CD pipelines in place, we evaluate the different cases on their performance. Because the focus of this case study lies on the integration process of DAST techniques and not on the accuracy or detection rates of the individual tools, we only provide an overview of the execution times of each pipeline.
4) Finally, challenges that we encountered during the integration process are listed in the results (Section IV). The goal of listing these challenges is to provide insights to DevOps teams to prepare them for what they can expect when they want to automate DAST using CI/CD. In Section V we provide solutions to the aforementioned challenges.

C. Tool Selection

This section describes the tools that we used in our case study. The main functional requirement for the selection of the tools is a command line interface (CLI), as a minimum, that can be used to control testing activities such as triggering attacks or to configure testing components. More beneficial are tools that can be directly addressed through code via an API using HTTP requests or client libraries. User Interfaces (UIs) are not required, but may be beneficial for creating test cases during development.

Automation of WAST: For WAST, we employ *OWASP ZAP* web security scanner, a versatile open source tool that can be configured for multiple types of security tests. It is also recommended by the Open Web Application Security Project (OWASP) foundation[17]. *ZAP* is a standalone application that is accessible via GUI, CLI, REST API and various client libraries. It comes with pre-installed known attack scenarios that are executed during its active scan. This is complemented by a spider scan that attempts to automatically discover entrypoints in a web application that these attacks can be performed against. The result is a highly automated test that requires little configuration. However, these basic scans are rather limited because customized HTTP POST requests are not supported by *ZAP* in this configuration.

Automation of SAS: Besides the execution of predefined attack scenarios, ZAP can also be configured to act as a proxy between a testing tool and the target application. In this proxy configuration ZAP will not make its own requests so an additional tool is needed to perform the actual attacks. We employ *JMeter*, a command-line tool that can perform parameterized requests against an API, as the testing tool. Using JMeter's GUI, one can easily generate the XML files that define a test case. JMeter cannot detect vulnerabilities and therefore ZAP is inserted as a proxy between JMeter and the target application. If required, JMeter can be configured to include malicious payloads or to perform fuzz testing.

Automation of BDST: ZAP can be used in combination with a BDD framework in order to perform high level use cases such as signing up, uploading files or filling in multi-stage forms. An example of such a framework is *SeleniumBase* [18], a wrapper for *Selenium*, which mimics user behavior to automate security testing for the aforementioned use cases. A convenient way for developers to define the test cases for BDST is the *Selenium* IDE *Katalon*. It allows the recording of user activities on the web application (e.g clicking a button) and converts them into an executable Python file. *SeleniumBase* executes these tests through the *Pytest* framework [19]. Similar to SAS, ZAP is inserted as a proxy between *SeleniumBase* and the target application.

IV. Results

In this section we present the results that are derived from our research process as described in Section III. The results are divided into the requirements for security testing in CI/CD (Section IV-A), a description of the concept of implementing dynamic security testing into CI/CD (Section IV-B), the performance of our approach (Section IV-C, and the challenges that we encountered (Section IV-D).

A. Requirements for DAST in CI/CD

Before discussing the integration of the dynamic testing techniques into automated CI/CD pipelines, we define the following requirements. The requirements are based on the commonly known DevOps requirements such as the ones described in [20], [8], [21] with certain extensions to meet DevSecOps goals.

R.1 **Quick build times** - Ensures that dynamic security testing is practical and every commit build takes no longer than 10 minutes to allow quick build fixes.
R.2 **Parallel pipeline jobs** - The pipeline should be able to run unit, functional, integration, security, etc. tests in separate jobs such that they can be run in parallel. This will speed up pipeline execution. This should include security tests at multiple abstraction levels.
R.3 **Testing of multiple versions** - The pipeline should be able to test multiple versions simultaneously (i.e. different branches or commits) without these pipelines interfering.

R.4 **Test every Commit** - Every commit to the remote repository should trigger a pipeline process. This ensures together with R.1 that vulnerabilities are detected early such that broken builds can be fixed quickly.

R.5 **Only build what is necessary** - Ensure that pre-built images for pipeline and testing components can be used for components that do not require frequent updates. This reduces overall run-time of the pipeline because slow builds do not have to be repeated every time the pipeline is started.

R.6 **Flexible deployment strategies** - The pipeline should provide a method to configure the deployment strategy being used. For some systems it may be desirable to deploy even with some minor vulnerabilities, whereas other systems should definitely not be deployed if any vulnerabilities are found. This allows DevOps teams to customize the pipeline to their project's needs. Other deployment strategies include selecting which tests are being executed at specific stages of the CI/CD process through the use of test scopes. It may for example be desirable to run a larger but slower set of tests before a system is deployed to production.

R.7 **Report vulnerabilities** - The system should not only report whether a pipeline job passes, but also provide clear test results in case of a pipeline failure. Specifically, security tests should report about detected vulnerabilities during testing. Clear reporting helps developers to quickly locate and fix the issues.

R.8 **Flexibility of testing technology** - The system should allow flexible integration of DAST tools or frameworks that are best suited for security testing of the specific application. Having the ability to select different tools for different projects allows DevOps teams to reuse the knowledge that has already been acquired by the team.

B. Concept & Implementation of DAST in CI/CD

In this section, we provide the details of our implementation of the dynamic security testing approach presented in the previous section. In addition to the tools that help to automate security testing, several other technologies were used in order to provide a test environment that can be executed on any CI platform. In order to build, execute and share the required applications involved in the particular test cases, we used the virtualization software Docker[3]. Docker enables us to containerize every application and run it e.g. locally or remotely in the CI environment. Further, the Docker Compose tool allows to create and combine multiple containers which is important to connect the different testing tools with the test application. For the CI environment we decided to use GitLab CI. The usage of this cloud platform is not only free but also provides CI/CD pipelines as a service[4].

Finding a way to test the integration of automated DAST in CI/CD is challenging because it requires an adequately sized web application in order to make statements about the different security tests. Since it is out of scope of this study to develop such an application with specific vulnerabilities, OWASP WebGoat was chosen as a target application. WebGoat is a deliberately insecure application that was designed for educational reasons on the one hand and for testing security tools on the other hand. We decided to use this open source project because its 89.100 lines of code represent a reasonably sized web application. In addition, the WebGoat community already provides a WebGoat docker image which can be pulled directly from the public repository. Furthermore, the documentation of WebGoat includes information about its vulnerabilities, such as SQL injection (SQLi) vulnerabilities, and how to exploit them.

In order to integrate DAST into CI/CD, a requirements first approach is used to identify where in the pipeline it would be beneficial to perform automated security testing. As was discussed in section II-B, the first stage is Continuous Integration. Unit tests are executed following the build of a new version in order to verify its implementation. From R.4 it follows that security tests should be run in addition to this. Therefore, the build stage is extended to also build the tools required for security testing. Because this step is executed after every commit that is made by developers, it is important that this stage is fast. Both SAS and BDST tests can be distributed among multiple jobs, while WAST testing is sequential due to limitations in ZAP and therefore WAST is excluded from this stage. Only running these faster tests contribute to satisfying R.1 and R.6.

When the pipeline advances to the continuous delivery stage, the system is deployed to a testing server. New vulnerabilities can be present due to a change of the environment and infrastructure, and therefore it is desired to run all prior tests again. Because the continuous delivery stage is only reached after the development team considers a version as complete, we can run the slower WAST tests here for better test coverage. This coverage is improved because each tool has its own strengths and weaknesses, thus running multiple of them has a better chance of catching vulnerabilities that were missed by earlier tests.

As a final step, the system is deployed to a production environment. Because DAST is able to test a live system, it is possible to do one final sanity check by running all security tests against the live system. If any vulnerabilities are detected, this is reported back to the development team which can then revert the changes quickly through gitlab its interface.

In order to satisfy R.2 and R.3 it is necessary to split each set of tests into separate jobs. This allows the testing jobs to be distributed among multiple runners. This then leads to more flexibility because more testing methods can be added in parallel jobs without affecting pipeline performance, also contributing to R.8. Gitlab-ci only provides a single docker-registry to use when transferring images between jobs. This limitation requires the use of tags in order to safely store multiple docker images. In order to ensure a unique tag for each image, a mnemonic name for a job is concatenated with

[3]https://www.docker.com

[4]https://docs.gitlab.com/ee/ci

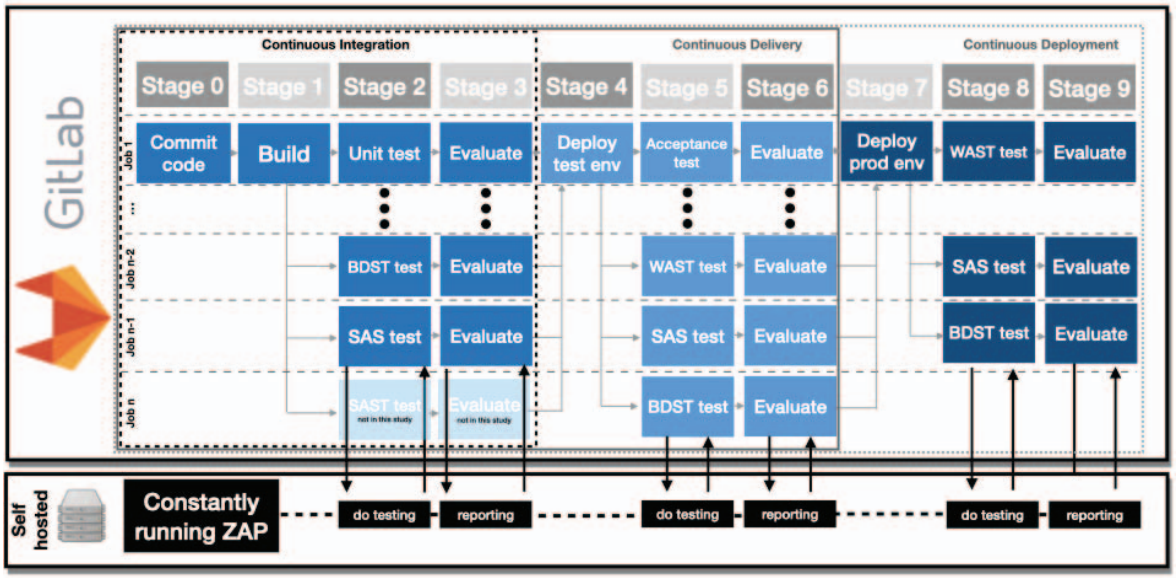

Fig. 4. The different stages of a CI/CD pipeline with the emphasis on parallel test execution of the various security testing techniques.

the unique commit hash that is made available to each instance of the pipeline, thus creating a unique identifier for each testing method.

Because ZAP acts as a proxy between the testing tool and the service under attack, it cannot simply return a test result to the testing tool. Instead, ZAP aggregates these results while analyzing traffic, and exposes these through its API. Therefore, after testing is done an evaluation stage is added to the pipeline that retrieves these results and uses them to decide whether the pipeline should pass or fail. Vulnerability thresholds can be configured in a *strategy.json* file. If any vulnerabilities were found, the evaluation component will print the test results to the standard output of the pipeline, thus contributing to satisfying R.7.

Because the primary focus of this paper lies on dynamic testing, no implementation for Static Application Security Testing (SAST) has been added, but it is still added for completeness to the pipeline in Figure 4 indicated in light-blue. The benefit of SAST is that the testing process is fast and easily integrated and can therefore be added in parallel to DAST and unit tests.

Finally, a novel idea is introduced to speed up the overall execution time of the pipeline. Because ZAP is an external component that does not get updated frequently, we can separate it from the rest of the pipeline and host it as an external service. This requires two-way communication between ZAP and the the pipeline, which is not natively supported by gitlab-ci. It is however possible to add a container to gitlabs runner that acts as a router to facilitate this communication. With the ability to run ZAP as an external service, it does not have to be rebuilt for every instance of the pipeline and thus we only build what is necessary, contributing to R.5.

Building of docker images is executed in the *build* stage of the pipeline. Each testing approach requires two docker images to be built, resulting in a total of 6 images being built. In order to speed up this process, each build is executed as a separate parallel job. A build job consists of three steps: First, a login to a remote docker repository is required. Secondly, the docker images are built and and tagged using the commit hash of the current branch. This allows us to push images multiple branches to the same repository without them interfering with each other. Finally, the images are pushed to the remote repository so they can be used in the next stage.

The second stage is testing, where the images from the previous stage are pulled from the remote docker repository and security tests are then executed. This stage uses a docker-compose in docker image[5], allowing us to run the test setup exactly as one would do on a local machine. Because the images must be pulled from a remote repository, a custom python script is used to update the *build* section of the compose file with an *image* entry pointing to the remote repository. Results are written into a volume that is shared with the CI pipeline. The contents of this directory are exported as GitLab CI artifacts.

Tools such as PyTest and JMeter require a running application to test. Since docker-compose's container dependency feature does not wait for a web server to be ready, this had to be implemented manually. For this the *wait-for-it.sh*[6] was used to delay starting the testing process until all dependencies are ready to respond to requests.

ZAP and WebGoat are web services that keep running until they are explicitly stopped. This means that without sending a shutdown signal, the CI/CD pipeline will never terminate. Therefore, at the end of the testing step a shutdown command

[5]https://hub.docker.com/r/docker/compose/
[6]https://github.com/vishnubob/wait-for-it

is sent to ZAP through its API. This results in a graceful termination of ZAP. However, WebGoat does not provide such an endpoint and therefore a different approach is needed. In order to ensure that WebGoat terminates after testing, the testing tool's image has been constructed using the same docker-compose in docker image as the CI pipeline itself. In order to make this work, a volume must be mounted into this image to make the pipeline's docker daemon socket available to docker inside the testing tool's image. After test execution the testing tool is then able to call the *docker kill* command. This will cause WebGoat to terminate once testing is complete.

The aforementioned artifacts that were exported in the previous stage are imported into an evaluation step. This last step then executes a python script that reads the JSON output from ZAP, and uses this to decide whether the pipeline should pass or fail. If a pipeline should fail, it is sufficient to exit the script with a non-zero status code. For an actual production setup, this step would then be followed by a deploy step. This deploy step will only be executed if the evaluation script exits with a zero as status code.

Integration of WAST in CI/CD: For the WAST integration we used the containerized ZAP and WebGoat containers. In order to control the tests, we added another container containing a simple script written in Python. The script makes use of the Python ZAP client library. With this library, one can easily control pro-active scans of ZAP. It receives the URL address of the WebGoat application as an argument to trigger spider and active scans in ZAP. This setup allows a complete scan of the WebGoat application. After the scans are finished, the aforementioned methods are used in order to terminate the docker-compose setup and evaluate the test results.

Integration of SAS in CI/CD: For the SAS test scenario we used again the containerized ZAP application for detecting malicious HTTP traffic. However, ZAP is now used in proxy mode and hence only forwards all traffic to the target application and analyses the responses. As was explained in Section II-A, we used JMeter to perform specific API scans. Therefore, we installed JMeter and the `.jmx` files in a docker container. In order to execute the tests one has to add entrypoints to the container that takes arguments and forwards them to the JMeter CLI inside the container. Thus, we can dynamically specify test files and setups ZAP as the proxy. This is important to detect security issues within the HTTP communication initiated by JMeter.

Integration of BDST in CI/CD: For the BDST technique we applied the SeleniumBase framework, which is installed in its own docker container together with two test cases. The first test case registers in the WebGoat application and the second uses the credentials created to log in and perform an SQLi attack. Similar to the JMeter docker container we needed to add an extra docker entrypoint in order to start SeleniumBase via the docker-compose command section. Because the SeleniumBase configuration refuses to accept the default docker-compose generated host names to configure ZAP as proxy (docker-compose gives random IP addresses to each container which can then be accessed using a mnemonic host name), we had to add a customized docker network to assign static IP addresses to each container to configure scanning for vulnerabilities. Through these static IP addresses, SeleniumBase is able to communicate with WebGoat using ZAP as a proxy.

C. Performance

We made a preliminary analysis of the extended CI/CD pipeline. The results for every test for each CI job can be downloaded from the GitLab CI interface. All three security tests have detected vulnerabilities [7]. The detection of these vulnerabilities causes the evaluation step of the pipeline to fail as intended. A single CI job, covering all three test scenarios, takes 14 minutes and 6 seconds. This run-time includes building of all components, starting the applications, performing the tests, evaluating the results of all tests, and exporting the artifacts containing the detected vulnerabilities. Subsequently, the three test techniques will be discussed individually. All setups used the WebGoat application as a service under attack, which is deployed using an already existing docker image. Therefore, it has no building time.

WAST included three docker containers. The building time for ZAP container is 6 minutes and 52 seconds and the container used to control ZAP requires 1 minute and 32 seconds. The execution of WAST takes 6 minutes and 4 seconds in total. The spider scan identified 13 resources along the path `http://webgoat:8080/WebGoat/` and the active scan detected 15 vulnerabilities. ZAP categorized those vulnerabilities by risk which results into 7 "Informational", 6 "Low", 1 "Medium", and 1 "High" risk. The vulnerability with the high risk was detected on the address `http://webgoat:8080/WebGoat/register.mvc` and denotes this resource to be vulnerable against a SQLi attack. Finally, the evaluation of this test technique took 1 minute and 8 seconds.

The SAS test setup also requires three components. The build time for its two build components are: ZAP in 6 minutes and 9 seconds and JMeter in 2 minute and 2 seconds. The run-time of the test takes 2 minutes and 31 seconds. The test was performed against `http://webgoat:8080/WebGoat/login` and ZAP detected two addresses that are vulnerable. The first address is exposed to a two "Low" risk vulnerability and one "Informational" risk. The second address is liable against 1 vulnerability which is a "Low"risk.

The BDST setup needs to build two containers, namely ZAP and SeleniumBase. The first component takes 6 minutes and 8 seconds to build and the second 4 minutes and 31 seconds. The duration of the test stage is 3 minutes and 45 seconds. During the two performed behaviour driven tests, ZAP detected 32 vulnerabilities, composed by 28 "Informational" and 4 "Low" risk security issues. Interesting is that one of the test cases included an SQLi attack where user passwords were exposed

[7]All results are derived analysing this CI job are available at: https://gitlab.com/rvbuijtenen/continuous-security/pipelines/128935397

TABLE I
VULNERABILITIES DETECTED FOR EACH AUTOMATED DAST TECHNIQUE

Test Type	# Tests	Inform.	Low	Medium	High	Total
WAST	13 URLs	7	6	1	1	15
SAS	1 URL	1	3	0	0	4
BDST	2 UCs	32	28	0	0	50

inserting SQL commands into the username field. This attack was not detected by ZAP. The evaluation of the test results took 1 minute and 5 seconds. An overview of these results can be found in Figure 5 and Table I.

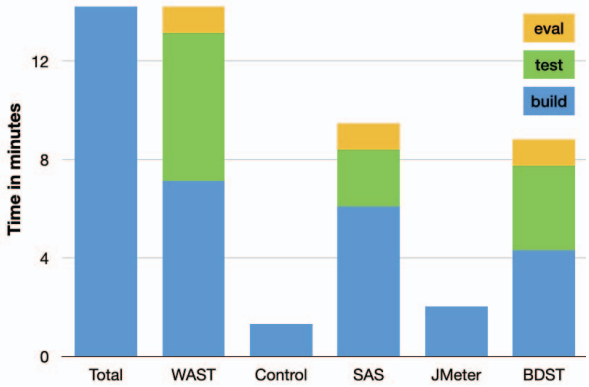

Fig. 5. Build-, test-, and evaluation-time for all pipeline stages

D. Challenges

In contrast to the quantified results of the case studies, we also present qualitative results because they are important to identify challenges in the integration of automated DAST into CI/CD. Solutions to the problems that are discussed here are presented in Section V.

a) Synchronization: In the docker-compose setup of all three testing techniques, we encountered the problem that all containers were marked as ready but the application inside the container was still starting. This resulted in tests being triggered while e.g. WebGoat or ZAP were not yet ready. For SAS this caused the program to exit without any test results, while for BDST this caused the SeleniumBase container to crash with an error.

b) Pipeline Termination: Another recurring problem was pipeline termination. Despite the tests finishing as intended, the remaining docker containers were still running. This is not a surprise because ZAP and WebGoat are standalone software systems that are designed to run until they are stopped explicitly. If this is not done the CI job will never finish and one could never determine if the dynamic security test has passed or failed.

A similar problem related to containerization was to get the results form ZAP. Since zap provides its results through a web UI, there was no clear way to extract these from the container. However, it is possible to make an HTTP request that downloads the test results in JSON or HTML format.

c) Configuration Issues: Another problem occurred with using SeleniumBase as a BDD framework. SeleniumBase can be configured to redirect requests through a proxy which works fine in native installations. However, SeleniumBase only accepts an IP address as a proxy target. Because docker-compose assigns a dynamic IP to a container when it is started, it is not possible to refer to this IP using the default configuration, hence further customization is required.

Using SeleniumBase we defined a test case that performed an SQLi against the WebGoat application. However, it turns out that the testing configuration is setup between the testing application and WebGoat's UI, rather than between WebGoat and its (backend) API. This resulted in ZAP not detecting the presence of leaked information because the leak is outside of the scope of what SeleniumBase is able to test.

V. DISCUSSION

As the results in Section IV show, all three testing methods can be performed in our setup and vulnerabilities are detected by employing the existing tools for test automation in a feasible way. The evaluation of the results stops the pipeline and thereby prevent the undesired deployment of security flaws to a production system.

One can easily see that the tests detected several security issues that were categorized on a low or even informal risk level (Table I). Depending on the scope of the system, the evaluation of the ZAP results can be configured in such a way that those alerts are ignored or only reported but do not lead to a pipeline failure. Finally, we could show that our approach is capable of satisfying the requirements which were defined in Section IV-A. In what follows, we will discuss the challenges that we encountered in our case study.

One demanding challenge is to keep the run-time of a pipeline to a minimum. In our case study we were initially not able to achieve the maximum execution time of 10 minutes. However, in our approach we suggest means to resolve this problem to a certain extent. Building the security testing component is the slowest part of our CI/CD pipeline. As already mentioned, we excluded this component from our pipeline and deployed it separately. This results in the desired reduction of the overall run-time and derives to a result that we consider to be adequate as it meets our requirement for quick build times. Another solution is to execute different testing types but also individual test cases in parallel. Note however that this applies only to SAS and BDST. Our WAST setup provides no built-in functionality that allows for distributed testing. Furthermore, the run-time depends heavily on the scale and complexity of the system that is being tested. If a WAST scan takes longer than what is considered as an acceptable waiting time for regular development, we recommend to only execute this type of testing for the continuous delivery and continuous deployment stages of the project.

The second and most challenging part that we encountered was the advanced expertise in containerizing all components involved in the test environment. Generally speaking, we found that many pitfalls in this area come from the isolated nature

of containerized applications and therefore a fair amount of knowledge of tools like Docker and GitLab CI are required. This included long starting times of components, termination of endless running containers, extracting test results, and mismatches of dynamic IP addresses. Development Teams should therefore consider to invest into advanced training for developers regarding containerization. The four problems of the containerization challenge are listed as follows:

1) Several services are executed before other required services or even the system under test are started. One can address this issue with adding synchronization means to the affected containers (mostly waiting for all services to be properly started). The result is that the testing container is paused until all services are available which solves this issue.
2) The CI/CD pipeline does not terminate due to web services that wait for an explicit shutdown. This can be solved by adding several shut down mechanisms. Solutions for this depend on whether a certain service already provides such a mechanism that merely has to be triggered or whether terminating an entire container needs to be forced.
3) Storing test results of stand alone tools in a CI/CD pipeline before a certain container is terminated is another problem related to containerization. We solved this problem by storing the test reports temporarily to disc. Subsequently, one needs to export those to the corresponding CI tool (artifacts in GitLab CI) in order to make them available to the development team.
4) The last problem is the default absence of IP addresses in container orchestration (e.g. docker-compose). Nonetheless, several tools require these addresses in order to properly perform their tasks. Therefore, one needs to introduce fixed IP addresses in their container orchestration.

The third challenge is to deal with increasing complexity of security tests. Especially for those techniques where the developer has to create a test case manually as it is in SAS and BDST, the number of tests will grow rapidly over time. Therefore, we suggest to consider SAS already in the API design. The team should fall back to the experience of a security expert in order to determine possible attack scenarios against this API. The experts should then be included in the test design as well. Subsequently, the API development should follow test driven development (TDD) and start with creating the SAS test case. This ensures that no API is forgotten and no vulnerabilities remain undetected.

Finally, no single testing technique is a silver bullet for detecting all security flaws. Hence, development teams have to tackle the challenge of using different testing techniques to cover as much vulnerabilities as possible. For example, the WAST technique requires the least amount of integration effort. The default setup of ZAP is however not capable of finding all resources since the spider scan is a.o. not capable of detecting resources that require authorization [13]. On the other hand, BDST allows testing from the perspective of a hacker. The setup that is suggested is capable of performing these scenarios as our two test cases show. However, we were not able to detect the SQLi attack scenario. This is not a surprise because ZAP is only a proxy between the BDD framework and the web application. The malicious request however is sent between the web application and its underlying web service. In order to detect those security flaws we suggest to use SAS in addition to BDST to analyse the requests sent to the web service's API. The SAS testing technique is the most flexible because it allows to test every single endpoint individually. This is important as was shown by the previous example of BDST. However, creating and managing tests for every single endpoint of an API can become increasingly complex for large applications. Furthermore, the flexibility of the tests can easily lead to forgetting certain aspects in the tests. Unfortunately, the solution to cover as much vulnerabilities as possible exacerbates the challenges of the test complexity. Furthermore, increased test complexity and maintenance for larger systems is already a known issue in DevOps, and therefore not a challenge unique to DevSecOps.

TABLE II
OVERVIEW ON INTEGRATION CHALLENGES

4 Challenges and proposed solutions of integrating DAST into CICD
1. Challenge - keeping the run-time at a minimum
• parallelize different testing types and if possible also individual tests • deploy testing tools such as ZAP outside the CI/CD pipeline • exclude testing techniques with longer run-time from CI stage
2. Challenge - lack of containerization expertise testing tools
• provide team training in containerization techniques
3. Challenge - test complexity
• apply TDD techniques for API design • integrate security experts in all development stages (follows sharing pillar of CAMS)
4. Challenge - vulnerability coverage
• combine testing techniques to achieve a higher coverage of vulnerabilities

VI. RELATED WORK

With the increased adoption of CI/CD pipelines in software development, the concept of DevSecOps has gained popularity in the research community. Many of the recent works have been in the form of surveys that try to define the core concepts in DevSecOps and provide perspectives of different stakeholders.

Myrbakken and Colomo-Palacios aim to provide a definition for DevSecOps, what its main benefits are and how the need for DevSecOps emerged from DevOps [2]. The authors found that DevSecOps is defined as the integration of security processes and practices that are meant to shift the mindset of all participants in the SDLC to get everyone to do what they can to ensure security of a system. Our work investigates the pitfalls of integrating security processes into the SDLC.

[8] presents a study in which six software developers were interviewed in order to get a better understanding of their view on the four pillars of DevOps: culture, automation, measurement and sharing. We concentrate mainly on the

automation pillar. In addition, our work provides preliminary solutions on how to increase the automation level in security testing.

Yasar and Kontostathis provide the 8 best practices on how to ensure sufficient security in DevOps [6]. These practices aim to deal with the negative feelings that developers have towards information security while being easily integratable into the rapid release cycles that are enabled by modern DevOps. This is achieved by shifting security from following a set of rules and guidelines to a proactive approach where security can be tackled by using creative solutions to solve though security problems at an early stage in the SDLC. However, none of these works present an implementation level case study and discuss technical challenges as described in this paper.

[22] presents an industrial case study to identify the challenges and best practices in adopting DevSecOps. Their work considers the challenges at the business process level (as opposed to implementation level) and is mostly tailored to separation of duties in performing tasks. Although we also identify challenges in adopting DevSecOps, our work focuses on integrating technical solutions into the SDLC.

Perhaps one of the most relevant work in terms of automated security testing in CI/CD is [13]. The author lists the scope and challenges found in the automation of security testing. Dynamic penetration testing and fuzz testing are discussed using practical examples with a number o tools such as OWASP ZAP, JMeter and Selenium. These tools are evaluated in three case studies on the security of web applications. Although interesting, the information provided are often incomplete and failure prone. In addition, the author discusses various security testing techniques in the context of CI/CD. However, he misses to identify relevant challenges one need to tackle to properly integrate DAST techniques in CI/CD pipelines.

In this paper, we define general requirements for dynamic security testing in CI/CD, identify challenges and provide solutions for addressing these requirements and challenges.

VII. Conclusion & Future Work

In this paper, we studied the integration of continuous (dynamic) security testing into CI/CD pipelines. To our knowledge, our work provides the first academic view on the topic. We defined eight requirements for a proper adaptation of automated dynamic application security testing for DevSecOps teams. These requirements ensure practical and agile development of web applications, web services and alike. In order to identify the practical challenges in meeting these requirements, we performed a case study by integrating three commonly known security testing tools into a CI/CD pipeline. We believe that the interested DevSecOps teams can benefit from our work as they can use our approach as a reference architecture for dynamic testing in CI/CD pipelines and learn from the challenges/solutions we outlined.

As future work, we want to focus our research on automatically patching detected vulnerabilities and automated test generation in case of behavioral changes in the application. First promising approach for the latter challenge is described by *Shoshitaishvili et al.* in [23]. We are planning to apply their methodology into our framework.

References

[1] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing," Gaithersburg, MD, USA, Tech. Rep., 2011.

[2] H. Myrbakken and R. Colomo-Palacios, "Devsecops: A multivocal literature review," 09 2017, pp. 17–29.

[3] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 25, 07 2015.

[4] J. Willis, "What devops means to me," https://blog.chef.io/what-devops-means-to-me/, 07 2010, accessed: 26-02-2020.

[5] J. Humble and J. Molesky, "Why enterprises must adopt devops to enable continuous delivery," vol. 24, pp. 6–12, 08 2011.

[6] H. Yasar and K. Kontostathis, "Where to integrate security practices on devops platform," *International Journal of Secure Software Engineering*, vol. 7, pp. 39–50, 10 2016.

[7] S. Kraemer, P. Carayon, and J. Clem, "Human and organizational factors in computer and information security: Pathways to vulnerabilities," *Computers & Security*, vol. 28, pp. 509–520, 10 2009.

[8] N. Tomas, J. Li, and H. Huang, "An empirical study on culture, automation, measurement, and sharing of devsecops," 06 2019, pp. 1–8.

[9] O. J. of the European Union, "Regulation (eu) 2016/679 - general data protection regulation," https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679#d1e1374-1-1, accessed: 05-09-2020.

[10] M. Kreitz, "Security by design in software engineering," *SIGSOFT Softw. Eng. Notes*, vol. 44, no. 3, p. 23, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3356773.3356798

[11] "Owasp top ten," https://owasp.org/www-project-top-ten/, accessed: 27-02-2020.

[12] T. Hsu, *Hands-On Security in DevOps: Ensure Continuous Security, Deployment, and Delivery with DevSecOps*. Packt Publishing, 2018.

[13] T. H.-C. Hsu, "Practical security automation and testing: tools and techniques for automated security scanning and testing in devsecops," 2019.

[14] P. Zech, "Risk-based security testing in cloud computing environments," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, March 2011, pp. 411–414.

[15] R. K. Lenka, S. Kumar, and S. Mamgain, "Behavior driven development: Tools and challenges," in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, Oct 2018, pp. 1032–1037.

[16] S. A. I. B. S. Arachchi and I. Perera, "Continuous integration and continuous delivery pipeline automation for agile software project management," in *2018 Moratuwa Engineering Research Conference (MERCon)*, May 2018, pp. 156–161.

[17] "Free for open source application security tools," https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools, accessed: 10-06-2020.

[18] "Seleniumbase (https://seleniumbase.com/)," accessed: 16-03-2020.

[19] "Pytest (https://docs.pytest.org/en/latest/contents.html)," accessed: 22-03-2020.

[20] M. Fowler. (2017) Continuousintegrationcertification. [Online]. Available: https://martinfowler.com/bliki/ContinuousIntegrationCertification.html

[21] ——. (2006) Continuous integration. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[22] V. Mohan, L. B. Othmane, and A. Kres, "BP: security concerns and best practices for automation of software deployment processes: An industrial case study," in *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*. IEEE Computer Society, 2018, pp. 21–28.

[23] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna, "Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 347–362. [Online]. Available: https://doi.org/10.1145/3133956.3134105