Exploiting DevOps Practices for Dependable and Secure Continuous Delivery Pipelines[*]

Thomas F. Düllmann, Christina Paule, and André van Hoorn
Institute of Software Technology
University of Stuttgart, Germany

ABSTRACT

Continuous delivery (CD) pipelines recently gained wide adoption. They provide means for short and high-frequent development cycles in DevOps by automating many steps after a commit has been issued and bringing it into production. CD pipelines have become essential for development and delivery. Hence, they are crucial and business-critical assets that need to be protected from harm in terms of dependability and security. DevOps practices like canary releasing and A/B testing aim to improve the quality of the software that is built by CD pipelines while keeping a high pace of development. Although CD is a part of DevOps, the DevOps practices have primarily been applied to the artifacts that are processed but not on the pipelines themselves. We outline our vision of using these DevOps practices to improve the dependability and security of CD pipelines. The goal is to detect, diagnose, and resolve dependability and security issues in the CD pipeline behavior. In this paper, we outline our envisioned roadmap and preliminary results from an ongoing industrial case study.

ACM Reference Format:
Thomas F. Düllmann, Christina Paule, and André van Hoorn. 2018. Exploiting DevOps Practices for Dependable and Secure Continuous Delivery Pipelines. In *RCoSE'18: RCoSE'18:IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering, May 29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194760.3194763

1 INTRODUCTION

In today's agile software development and DevOps practices, automation is an important aspect in order to build, test, and deliver high-frequent feature increments [3, 6, 17]. Our understanding of DevOps is aligned with the definition by Bass et al. [3], namely "a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality". One important DevOps practice is the usage of continuous deployment or delivery (CD) which helps to automate as many steps from a source code commit to the deployment of a software artifact to production [3, 12]. In many cases, the core element in this context is a CD pipeline. It consists of stages that are executed based on a trigger (e.g., push to source code repository). Usually, stages have specific responsibilities (e.g., compiling the source code, executing unit tests). Every stage can contain multiple tasks that are executed step by step. In addition, they allow to achieve a high frequency of feature increments in fast increment cycles with fast feedback to the developers [12].

Due to the importance of CD pipeline tools (e.g., Jenkins), a new menace emerges that might endanger the whole development process. In case a CD pipeline does not work as intended or is not available at all, most advantages of agile and DevOps processes are at risk. Additionally, all steps that were automated for a particular reason would also not be available anymore which would impose an even bigger impact on the overall workflow.

This leads to the fact that the dependability and security of CD pipelines become existential properties for the software development process as for the software product itself.

Existing works focus on securing the software artifacts in a CD pipeline [8, 16] and increasing trust and security of CD pipelines by limiting access and permissions as well as using virtualization [2, 4, 10, 18].

To tackle this issue, we plan to apply DevOps practices like canary releasing, A/B testing and chaos engineering to the CD pipelines themselves for detecting, mitigating, and preventing dependability and security problems. To our knowledge, no work on applying DevOps practices to pipelines themselves for improving their dependability and security has been conducted.

The paper is structured as follows. The background and related work in the different fields are outlined in Section 2. Section 3 presents our vision and the planned steps we intend to follow. In Section 4 we present our evaluation and preliminary results. We conclude the paper in Section 5.

2 BACKGROUND AND RELATED WORK

A typical CD pipeline works as follows. It is triggered (e.g., by a push to the version control system), it downloads the source code from the source code repository and checks out the branch that is supposed to be processed. The repository also contains a pipeline definition file that defines the structure and the behavior of the CD pipeline (e.g., a Jenkinsfile), following the common practice of infrastructure as code [11]. By reading the pipeline definition file, the pipeline tool knows which stages and tasks have to be executed during the CD pipeline build. Afterwards, it compiles the source code. This also makes sure that there are no compilation errors in the source code. As a next step, unit tests are executed to confirm that all test cases are successfully completed. To make sure that the application as a whole is working, it is deployed to a local or cloud-hosted test environment. There it is closely monitored while put under test load to ensure that it works flawlessly. When this

[*]This work is funded by the Baden-Württemberg Stiftung (ORCAS Project, Elite Programme for Postdocs).

RCoSE'18, May 29, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5745-6/18/05…$15.00
https://doi.org/10.1145/3194760.3194763

succeeded, the application is ready to be deployed to actual live production systems. CD pipelines can be complex infrastructures, including components such as external services hosted by third parties, as well as infrastructure hosted locally or in the cloud.

According to Avizienis et al. [1], dependability is defined as "the ability to avoid service failures that are more frequent and more severe than is acceptable" comprised of the combination of several properties: *availability* (readiness for correct service), *reliability* (continuity of correct service), *safety* (absence of catastrophic consequences on the user(s) and the environment), *integrity* (absence of improper system alterations), and *maintainability* (ability to undergo modifications and repairs).

In parts, this also overlaps with the aims of IT security which cover the protection of assets that are represented by data that is stored or transferred electronically. The three main concerns in information security are covered by the so-called *CIA*-triad [7]: *confidentiality* (accessibility of data only by authorized individuals), *integrity* (manipulation of data only by authorized individuals), and *availability* (accessibility of data whenever it is needed).

Means for dependability and security are fault prevention, tolerance, removal, and forecasting [1].

The existing literature shows that the topic of dependability and security of CD pipelines is of relevance, though not researched in too much detail, yet. Although some works investigated the security of CD pipelines themselves, none of them did or planned to use DevOps practices for the detection, mitigation, or prevention of dependability and security problems.

Ullah et al. [18] identified security threats of CD pipelines by quantitative and qualitative analysis and propose five security tactics to improve the security of the CD pipelines. The identified threats are mostly related to the security of the pipeline facing the outside (i.e., the Internet). Therefore, their tactics focused on using container technologies (e.g., Docker) and applying their features by using fresh immutable instances of a service. Stažić [16] worked on integrating security in agile development processes as well as exemplary CD pipelines while focusing on the security properties of the artifacts. In our work, we are not focusing on the development processes but on the security of the pipeline itself. Bass et al. [2] applied security hardening to a typical deployment pipeline by limiting the access and permission of single components to increase the trust in these components. Kuusela [8] reviewed software security testing tools that can be executed in a continuous integration process and applied them in case studies. That work is focused on tools that are related to the security of the artifacts that are processed in a CD pipeline, while we intend to focus on the CD pipeline itself. Gruhn et al. [4] conducted a security risk analysis of public continuous integration services and identified possible attack vectors. In a proof of concept, they aimed to eliminate one class of attack vectors using virtualization. Lipke [10] conducted a threat modeling process for an exemplary software supply chain using Docker with the STRIDE approach. Furthermore, he identified that build servers and production environments are critical components.

We might adapt some aspects of the risk analysis and the threat modeling process of the two works above [4, 10], while in contrast to them, we plan to use DevOps practices to detect, prevent, or mitigate security and dependability problems of CD pipelines.

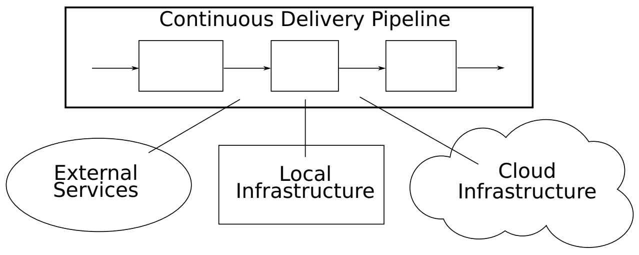

Figure 1: CD Pipeline Dependencies

3 VISION AND ROADMAP

The central goal of our approach is to improve the dependability and security of CD pipelines. To achieve this, we are envisioning to use DevOps practices. Particularly, we aim for approaches to detect, mitigate, or prevent problems that become even more problematic due to the fact that CD pipelines are business-critical infrastructures with many systems that depend on them. Figure 1 visualizes additional dependencies that CD pipelines can have and that might increase the risk of CD pipelines to be compromised, or the risk of other assets being compromised by the CD pipeline. These are *external services* that are connected to the CD pipeline (e.g., online code coverage analysis). Additionally, CD pipelines can interact with local infrastructure such as testing environments, worker nodes with specific machine configurations for the execution of builds, or even databases with confidential data that cannot be made publicly available. Also, cloud infrastructure can be connected to a pipeline — for example for worker nodes, deploying artifacts to test or even production environments.

In order to achieve our goals, we intend to follow the roadmap outlined below:

3.1 Identification of Typical CD Pipelines

As a first step, we intend to investigate definitions for CD pipelines to get a possibly complete overview. Apart from that, we are interested in finding out what typical pipelines in the industry are. We are especially interested in information about structure, behavior, used tools, and the means taken to ensure dependability and security. Then, we will search for existing approaches and tools to define, specify, or model CD pipelines and their properties. We envision to extend the model for CD pipelines by Wettinger [19]. In addition, we plan to investigate to what extent CD pipelines are different from other software systems that are business-critical to understand specific requirements CD pipelines might have.

3.2 Identification of Dependability and Security Threats Specific to CD Pipelines

In terms of dependability and security, we plan to research taxonomies that allow us to categorize different dependability and security aspects. To be possibly exact in classifying what threats we want to focus on, we plan to use existing taxonomies. Avizienis et al. [1] provide basic concepts and a taxonomy of dependable and secure computing. For example, their classes of combined faults provide a fine-grained collection of different aspects (e.g., when the fault occurred, what the intent of the fault introduction was). Langweg and Snekkenes [9] give a classification for malicious software attacks against applications which could provide another view of classification in addition to the previously mentioned taxonomy by Avizienis et al. [1]. These will be used as starting points for our classification efforts. Particularly, we intend to find possible existing approaches that model and analyze dependability and security aspects of CD pipelines.

Based on the results, we want to identify threats that are specific for CD pipelines and find out means to prevent, mitigate, or detect threats. This should eventually result in a focused threat model and a threat classification for CD pipelines. In order to assess security-critical aspects of CD pipelines, it is necessary to identify relevant threats. By applying threat modeling such aspects can be found using structured approaches [15]. The structured approach we intend to apply is the so-called STRIDE threat modeling approach. The name STRIDE stands for the different vulnerabilities that this approach intends to cover: **S***poofing identity*, **T***ampering with data*, **R***epudiation*, **I***nformation disclosure*, **D***enial of service*, and **E***levation of privilege*. Depending on the level of detail, each of these aspects is applied to the application, component, or feature to identify whether it could be affected by that aspect.

Table 1: STRIDE example table

Component	S	T	R	I	D	E
CD Pipeline Tool (Jenkins)	x	x		x	x	x

The key idea — as depicted in Table 1 — is to specifically focus on each of the aspects in the STRIDE approach and think about possible risks in that specific component. Based on this, one can mark the aspect as relevant by putting an 'x' in the table and continue from there by elaborating on the details in a textual description. As an example, we can define a possible weakness for the denial of service (the **D** column in the table) for the CD pipeline shown in Table 1: if the CD pipeline starts a build for every pull request that is created in a public version control system, an attacker could create many pull requests to saturate all workers that the project has available. This would lead to the problem that the CD pipeline would lead to a delay or even a complete halt of the development for that project and possibly all projects that also use the pool of workers.

3.3 Investigating the Use of DevOps Practices

Furthermore, we will investigate novel methods, tools, and practices that can improve the dependability and security of CD pipelines by using DevOps practices.

We plan to create proof-of-concept setups for the usage of DevOps practices (e.g., canary releasing, A/B testing) combined with thorough monitoring to evaluate their applicability.

At first, we plan to investigate canary releasing and A/B testing as possible practices from the DevOps context. Additionally, we want to use fault injection and monitoring as supplemental tools.

In order to detect problems, we plan to initially use baselines as applied in chaos engineering [14] and possibly adapt this approach.

Canary Releasing. In the context of DevOps, canary releasing stands for deploying new versions of a program on a small fraction of the production environment and redirecting a fraction of users to these instances with the new version [6]. While the canary instances are used, the system can be monitored to detect whether errors or other degradations occur. If errors occur, the redirection to the canary instances can be disabled and the instances be rolled back to the previous version. In case no errors occur, a rolling upgrade for the rest of the production instances can be triggered; either manually or automatically.

When applying this to the example CD pipeline, the pipeline as a whole or parts of it could be executed in a sandbox while monitoring its behavior. In case it would show major changes compared to the behavior this could trigger an alert. Instead of deploying the artifacts to a test environment in the cloud, it could be deployed to a sandbox environment.

Our intention in applying the canary releasing practice to CD pipeline execution is to assess the behavior before possible problems affect the actual production artifacts. This could be achieved by executing builds in a confined environment with detailed monitoring enabled. Due to the separate execution context, the required resources add up to the resources needed for the regular pipeline execution. The advantage of this approach is the possibility of mitigating problems before they have an impact on production systems or artifacts. This approach might not be practicable for pipelines that are very complex or have external dependencies as the creation of a corresponding confined execution environment might be very challenging.

A/B Testing. In contrast to canary releasing, A/B testing is used for comparing different designs, approaches, or concepts [3]. For example, one group of users is presented with variant A of a website and the other group is presented with variant B of the same website. In the end, metrics (e.g., number of sold items) for both variants are compared to decide which design is better suited.

When applying this to the example CD pipeline one could either execute the whole pipeline or specific stages with different configurations in parallel. That way it would be possible to compare the behavior to see what impact some changes would have. For example, one could execute one unit testing stage with the old configuration and another one with an altered configuration in parallel to see whether tests are affected by the change.

In our intended application of A/B testing, we plan to execute pipelines with different configurations with the same input to see how they perform in comparison to be able to find the best setup and identify possible issues affecting dependability and security.

Fault Injection. In the context of DevOps a new field of fault injection in live experiments is emerging [14]. The so-called chaos engineering injects faults into production systems to evaluate whether a system is resilient against these kinds of faults. Apart from faults, also failures can be injected to simulate the failure of parts of the system. In the context of CD pipelines, fault injection can help to identify dependability issues when components fail. An example based on Figure 1 could be, that the failure or degradation of an external service could be simulated to assess the dependability of the CD pipeline.

Monitoring. In order to be aware of issues in a system, monitoring is of utmost importance [3]. Especially in dynamic environments, monitoring is a helpful tool to get insights in the actual state and behavior of a system. Also, according to the Open Web Application Security Project (OWASP) [13], insufficient monitoring and logging is one of the top ten security issues. In case monitoring and logging is not present, this allows attackers to probe and attack systems without being detected.

In CD pipeline tools (e.g., Jenkins) it is usually easy to see the differences of errors and warnings from quality assurance tools (e.g., CheckStyle, PMD, JUnit) in comparison to previous builds. Though, apart from the time a specific stage or task took to execute, there is no information that may give insight whether the behavior of a stage or task changed (e.g., in terms of CPU utilization, memory consumption). We plan to take these measurements into account to find anomalous behavior when a new build is executed. Additionally, thresholds could be implemented that let the stages fail in case they are violating the thresholds set in comparison to previous builds. The advantage of this approach is that — apart from the extended monitoring — no additional resources are required. As this monitoring takes place during the pipeline execution, it is unlikely that problems will be mitigated. This approach could only be capable of detecting possible anomalies.

4 EVALUATION AND PRELIMINARY RESULTS

For the evaluation, we plan to do lab experiments and conduct industrial case studies with real-world CD pipelines.

We are currently conducting a first case study with an industrial partner, for which we can report preliminary results. Particularly, we focus on the identification of dependability and security threats, as outlined in Section 3.2. The industry partner provides insight into representative CD pipelines used in their software development process. The aim is to identify potential weaknesses and investigate them in detail depending on the level of risk they impose on the CD pipelines. We followed a two-fold research method. First, we conducted a qualitative questionnaire-based survey among developers to get an impression of the awareness of security risks for CD pipelines and current practices to address them. The results show that even though many of the developers are involved in the usage, configuration, and installation of CD pipelines, most of them are not aware of the business-critical risks that a CD pipeline imposes on the software development process. Second, we apply the STRIDE approach (Section 3.2). At the same time we use the case study to identify characteristics of typical CD pipelines (Section 3.1), as well as to come up with potential applications of DevOps practices (Section 3.3). Also, we plan to adapt concepts for creating lab experiment environments.

5 CONCLUSION

CD pipelines are not only important but critical for today's software development processes. We presented our ideas on applying DevOps methods — such as A/B testing and canary releasing — not only on the software artifacts that are produced and tested in a CD pipeline, but on the CD pipelines themselves. We are envisioning that this approach can support in detecting, mitigating, and preventing dependability and security problems in CD pipelines.

Though there are several prospected advantages, there might also be limitations regarding the applicability of such approaches. One of the possible hindering aspects is the vast complexity of CD pipelines. At some point it is very likely that a test execution may not be of any help if the CD pipeline has too many essential external dependencies. In case CD pipelines are utilizing a high amount of resources or are very time consuming during their execution, it might not be efficient to execute regular test runs. Another hindrance could be regulatory requirements that need to be overcome. According to Humble [5], the application of DevOps practices does not necessarily break such requirements but can possibly improve transparency and ease of implementation.

As a next step, we plan to continue to work on the industry case studies by investigating the details of the found weaknesses. Also, we want to research further ways of defining and modeling CD pipelines.

REFERENCES

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Sec. Comput.* 1, 1 (2004), 11–33.

[2] Len Bass, Ralph Holz, Paul Rimba, An Binh Tran, and Liming Zhu. 2015. Securing a deployment pipeline. In *Proc. IEEE/ACM 3rd International Workshop on Release Engineering (RELENG)*. IEEE, 4–7.

[3] Leonard J. Bass, Ingo M. Weber, and Liming Zhu. 2015. *DevOps — A Software Architect's Perspective.* Addison-Wesley.

[4] Volker Gruhn, Christoph Hannebauer, and Christian John. 2013. Security of public continuous integration services. In *Proc. 9th International Symposium on Open Collaboration (OpenSym)*. 15:1–15:10.

[5] Jez Humble. 2017. Continuous Delivery Sounds Great, but Will It Work Here? *Queue* 15, 6, Article 70 (Dec. 2017), 20 pages.

[6] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education.

[7] ISO/IEC. 2016. ISO/IEC 27000: Information technology — Security techniques — Information security management systems — Overview and vocabulary. (02 2016).

[8] Juha Kuusela. 2017. *Security testing in continuous integration processes.* Master's thesis. Aalto University, School of Science, Finland.

[9] Hanno Langweg and Einar Snekkenes. 2004. A classification of malicious software attacks. In *Proc. International Conference on Performance, Computing, and Communications (IPCCC)*. IEEE, 827–832.

[10] Simon Lipke. 2017. *Building a Secure Software Supply Chain using Docker.* Master's thesis. Hochschule der Medien, Stuttgart, Germany.

[11] Kief Morris. 2016. *Infrastructure as code: managing servers in the cloud.* O'Reilly Media, Inc.

[12] Michael Nygard. 2007. *Release It!: Design and Deploy Production-Ready Software.*

[13] Open Web Application Security Project. 2017. OWASP Top 10 - 2017. (2017).

[14] Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri. 2017. *Chaos Engineering: Building Confidence in System Behavior through Experiments* (1st ed.). O'Reilly.

[15] Adam Shostack. 2014. *Threat modeling: Designing for security.* John Wiley & Sons.

[16] Damir Stažić. 2017. *Security DevOps: Konzeption einer Umgebung zur Integration von Sicherheitstests in agile Softwareentwicklungsprozesse.* Master's thesis. Reutlingen University.

[17] Matthias Tichy, Michael Goedicke, Jan Bosch, and Brian Fitzgerald. 2017. Rapid Continuous Software Engineering. *Journal of Systems and Software* 133 (2017), 159.

[18] Faheem Ullah, Adam Johannes Raft, Mojtaba Shahin, Mansooreh Zahedi, and Muhammad Ali Babar. 2017. Security Support in Continuous Deployment Pipeline. In *Proc. 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. 57–68.

[19] Johannes Wettinger. 2017. *Gathering solutions and providing APIs for their orchestration to implement continuous software delivery.* Ph.D. Dissertation. University of Stuttgart, Germany.