DOI: 10.2507/33rd.daaam.proceedings.xxx

CI/CD TOOLSET SECURITY

Vedran Dakic, Jasmin Redzepagic &
Matej Basic

This Publication has to be referred as: Dakic, V[edran]; Redzepagic, J[asmin]; Basic, M[atej]; (2022).CI/CD toolset security, Proceedings of the 33rd DAAAM International Symposium, pp.xxxx-xxxx, V. Dakic (Ed.), Published by DAAAM International, ISBN 978-3-902734-xx-x, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/33rd.daaam.proceedings.xxx

Abstract

Modern application development has redefined the way teams develop their solutions. Instead of using their workstations for building code, a lot of teams have resorted to using microservices for CI/CD systems. CI stands for continuous integration while CD denotes continuous delivery. Combined these two things mean that using a CI/CD system code goes into building and testing cycle as soon as the developer submits it. Since the whole system is inevitably complex, almost all the systems are using a combination of technologies to manage both the building and delivery part as well as underlying services that make building possible. In this space technologies such as Kubernetes or OpenShift are becoming a norm.
From the security perspective this creates a whole new problem since such a system has to be deeply integrated into the core of the business network, and any potential threat to the CI/CD infrastructure is immediately a threat to the whole internal infrastructure. CI/CD tools need to have advanced privileges, they have to be able to access code repositories, user directories, complete development environments and even bare metal servers in order to optimize the delivery process. This means that attack surface in such a system is enormous and exploiting it means gaining access to large part of the business infrastructure.
Securing such a heterogenous system is a big task and, in this paper, we address most important challenges.

-Keywords: CI/CD; Kubernetes; Openshift; Gitlab; Jenkins; ArgoCD

1. Introduction

During the middle of 2010 decade there was a large shift when it comes to application development, especially in the web application space. Established monolithic building architecture involving dedicated servers and large server pools slowly became too complicated and too financially wasteful so a new solution was needed. Old model has an inherent problem of underutilization since a lot of resources are allocated on servers that perform either a single or small number of steps in the build process. Since this creates a large serial pipeline, this creates a lot of wasted time spent waiting for a task to finish. Virtual machines are a partial solution to this but only when containers and orchestration frameworks become available was a true solution possible. [1]

Containers enabled creating environments that contain only the objects and resources needed for a particular task and running a lot of tasks in parallel on same hosts. This meant that the hardware utilization went up while ability to manage software differences across platforms was simplified.
Continuous Integration (CI) and Continuous Delivery (CD) systems have an inherent problem with dependencies, every attempt to support many architectures and platforms means that at any given time there are a lot of similar but subtly different environments being used[2]. At the same time updating the environments in which the build process is happening is almost constant and, in this task, containerization enables probably the quickest way to update or redeploy whole environments.

Security in the context of a CI/CD system should therefore cover not only least privilege set, access control, monitoring and all the other best practices for the system itself, but we also need to track all of the components and their internals. This involves tracking changes in the source code, containers, all the tools and all the dependencies our system uses to produce the resulting application.

For all of the stated reasons CI/CD systems are a big attack surface from the security perspective and any improvement in the way security is designed in the build system itself has huge ramifications on the security of the final product. Our aim is to provide essential guidelines on how to both secure and restructure a typical CI/CD system to make it more robust.

2. Typical deployment scenario

How this all works in a typical development environment is relatively simple when abstracted enough. A developer usually manually develops and updates application code. In reality, this is mostly manual work, but sometimes rapid development tools assist in this part of the process. After a particular piece of code is finished it is submitted for review by the team. Review is usually done by a senior developer. Sometimes this step also involves automatic review by different tools that try to enforce standards in code syntax to keep the code uniform across the whole code base.
After review code is merged with the appropriate code tree at which point CI/CD system identifies code changes and starts build deployment. Depending on the particular configuration this may start a partial of complete rebuild of the finished product.

Next step in CI/CD is usually some sort of compliance and functional testing done automatically to ensure that most obvious errors are spotted. Once all this is done without errors code is packaged into test environment to be further tested by QA (Quality Assurance) team.

In this process CI/CD is the most important part of the infrastructure since it can access entire codebase, can introduce changes to the code, can deploy and run arbitrary code on any given part of the infrastructure and in the end can sign and deploy this code once application is assembled as a package.

All this makes CI/CD a prime target for any sort of attack since a successful attacker can disrupt the build process but also can directly inject code into the application and create even greater problems if this code injection ends up released as a signed and finished application. This kind of attack is sometimes called supply-chain attack since it tries to influence not the finished product but the production of the said product itself. If successfully done this kind of attack is complicated to detect and requires a lot of resources on response team side of things.

3. Best practices for CI/CD

CI/CD systems tend to be heterogeneous and comprise of different subsystems that are by themselves quite complex. Usually, we are talking about some sort of a code repository or code storage solution, Build / Deployment solution and some sort of orchestration tool as the core of the system. Rest of the system are different tools that are specific for each build. [4]

Name of the tool	**Bamboo**	**Gitlab**	**Jenkins**
Price (per agent)	Paid (from 10$)	Paid (from 4$/month)	Free
Type of deployment	OnPrem/Cloud	OnPrem/Cloud	OnPrem/Cloud
Free version exists?	Yes	Yes	Yes
Platforms supported	Windows, macOS, Linux, Solaris	Linux (Ubuntu, Debian, CentOS, Oracle)	Windows, macOS, Linux, Unix-alike

			operating systems

Table 1. Common CI/CD tools

Problem this poses from the security perspective is that we not only need to keep all the subsystems and tools updated, but we also need to keep inventory of their deployment and place in the whole system. Frequent updates and making sure we are not using outdated tools is the first thing that needs to be done. At the same time, we will sometimes HAVE to use tools that have known vulnerabilities due to their behaviour on a particular platform – a lot of applications we are developing can use older dependencies on older platforms and sometimes updates are no longer available. This is where inventory comes into play, every such dependency needs to be documented for our security team to apply additional measures to ensure safety.

Next thing to do is privilege reduction. Although the system needs to have broad permissions, every module has its own permissions that we can apply just to that module. For example, if there is a need for a module to read something from the database only read permissions must be given to that module ensuring nothing will end up being changed if there is a breach in that module.

Network connections are another part of the security. Every connection is a potential risk, but we can be quite strict in limiting access. For example, we can be sure which workstations belong to developers and assign them special access to the system. Same goes for containers that build and deploy the solution. IF a permission is not needed it shouldn't be given in the first place. Every access should be logged, and a monitoring solution used to ensure we are able to track all the actions and changes done to the system. [5]

This monitoring and access also should protect from inside attack attempts and should keep information compartmentalized if at all possible.

Next thing to monitor and pay special attention to are the credentials and the identification keys used across the system. A particular CI/CD can access not only the database but also other resources like cloud, both internal and external, remote shares, API endpoints, SSH endpoints, certificates and other sensitive data. Every piece of information should be encrypted, and keys automatically rotated to ensure security of data-at-rest.

Source repository is part of the CI/CD system but also has other uses inside the team. Since it will be accessed by the team directly, sometimes outside of the CI/CD scope, access should be monitored and limited.

Last thing that can be considered a best practice is making sure we track possible vulnerabilities to all of our tools and to be ready to apply updates as needed while also keeping track on possible interactions between subsystems that can be broken by updates. CI/CD systems tend to have a lot of custom scripts and a lot of knowledge is needed to make sure system runs after updates.

4. Practical examples of security

Automated testing should not only be limited to testing pure functionality of the code, but also should include security[8], [10]. Static code analysis is one such solution that enables us to detect potential security problems in the code. There are solutions available on the market that can deal with a lot of specialized vulnerabilities, for example there is a library called Selenium that enables us to automate SQL injection checks in the code.[6] Solutions like this enable "shift-left" security concept, trying to implement security as early as possible in the application lifecycle.

Developers are often using special in-code tags to mark certain parts of the code to be executed, updated, or disabled. If we can implement these tags into our checks this can help us identify parts of the code that needs to be checked or even disabled. This also includes checks for obsolete dependencies. We mentioned that these can be part of the development process due to compatibility reasons, but we should make sure that usage scope for such parts of code must be limited to a minimum.

Rollback procedures are also a big part of the security planning. Sometimes a vulnerability will pass all the checks and end up in the production application. We must have a feasible plan on how to restore to a previous working state without the vulnerability. This includes CI/CD system having older versions of application available if needed.

We mentioned the need to automate monitoring and logging of the CI/CD system. Production environments these days usually already have some sort of SIEM system implemented. Scope and reach of such system should be extended not only to application behaviour but also to the behaviour of the whole CI/CD system.

Our goal in all of this is to have complete control over CI/CD so we can spot not only changes in the code but also changes in the system that builds the code itself.[9]

Even having implemented all the recommendations outlined here we are just scratching the surface of security since we are talking about a complex, heterogenous system. In this paper we concentrated only on the most important aspects of securing the CI/CD environment, reader should understand that the security of the CI/CD system also depends on all the underlying components.

5. CI/CD security in Jenkins

When it comes to One of the most popular and often used CI/CD tools is called Jenkins [7],[3]. This tool is free and open source, so it is both highly transparent for the user but also a frequent attack vector. Jenkins tends to be primary target of a lot of attacks that then use other tools to spread horizontally across resources.

Being such a complicated and exposed tool, Jenkins has a robust security built in. A lot of the defaults are already strict out of the box, and update system is integrated into the tool itself. Jenkins also supports plugins, and they are in the scope of the update system. Unfortunately, Jenkins has seen hundreds of vulnerabilities published as public information so updating the system has to be frequent and complete.

Jenkins is structured in two parts – Jenkins server that enables all the functionalities of the system and Jenkins agents that are installed on build nodes, virtual machines or containers. Security of the server is usually in the hands of the administrator but since we are talking about a distributed system with high privileges a lot of functionality, especially scripting, is done by the users. Users are usually responsible for node configuration, script coding and deployment and everything else involving turning source into application itself. Since all this is executed by the agent there is a huge potential for security compromise.

Having agents also means that we can limit user interaction with the system – agents can be designated to specific projects so compromise of a single project can be isolated from other projects or the system itself.
Environment variables are one more thing we need to have in mind. Variables are used to pass different pieces of information between parts of the build process, most commonly build scripts. Variables can be divided by the phase of the build process they belong to (test, QA, production). This means we can keep sensitive data separate from the end product, but also that we need to keep track of variables at the same time.
Jenkins is a popular tool and a very powerful one, but it requires a lot of attention, and a dedicated team to keep it running. Main challenges in running Jenkins are updating the system while ensuring that the entire pipeline works after each update.

5. Conclusion

Modern development influences CI/CD systems and tools in making them more complex and advanced than ever before. Placing CI/CD in the core of the infrastructure and giving it control over almost every part of the system makes CI/CD security more important than ever. For this exact reason security has to be implemented from the moment we create any CI/CD infrastructure including not only the source control and build systems but also Kubernetes, OpenShift or whatever our system uses. Since CI/CD directly controls build process any intrusion not only compromises the system that we are working on but also the resulting application. All this means that security of the CI/CD process is probably the mose Security has to be layered, and implement all the standard security controls together with all the security practices for individual products that are used in the build system. Since systems tend to be heterogeneous and span different vendors, applying all the updates can prove problematic due to the way they can affect the system itself. Solution to this is to try and find the balance between using latest updates and making sure our system behaves as desired, which may also include unit tests that are performed while using the system on the system itself.

Human factor is also very important here since users are almost exclusively developers themselves and it can be difficult having them conform to different security challenge. But even when taking all the factors into account, in this particular case security has to be implemented since CI/CD is a core system that can not only compromise the work environment but also potentially can compromise the entire end product of the said system.

6. References

[1] Smart, J.F., Jenkins: The Definitive Guide: Continuous Integration for the Masses, , O'Reilly Media, 2011 ISBN: 9781449313654

[2] S. Garg and S. Garg, "Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security," 2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR), 2019, pp. 467-470, doi: 10.1109/MIPR.2019.00094.
[3] V. Armenise, "Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery," 2015 IEEE/ACM 3rd International Workshop on Release Engineering, 2015, pp. 24-27, doi: 10.1109/RELENG.2015.19.
[4] Seremet, Z[eljko] & Rakic, K[resimir] (2021). Best Approach to Security in Azure Devops, Chapter 18 in DAAAM International Scientific Book 2021, pp.223-230, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-31-0, ISSN 1726-9687, Vienna, Austria DOI: 10.2507/daaam.scibook.2021.18
[5] Kafol, C[iril] & Bregar, A[ndrej] (2017). Cyber Security – Building a Sustainable Protection, Chapter 07 in DAAAM International Scientific Book 2017, pp.081-090, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-12-9, ISSN 1726-9687, Vienna, Austria DOI: 10.2507/daaam.scibook.2017.07
[6] https://owasp.org/www-community/attacks/SQL_Injection, 2022, Open Web Application Security Project, OWASP Foundation, Inc. Accessed: 2022-6-30
[7] https://www.jenkins.io/doc/book/security/, Jenkins project, Software in the Public Interest, Inc. Accessed: 2022-6-30
[8] Securing Devops: Safe Services in the Cloud (Paperback or Softback); Vehent, Julien; 2018; Published by Manning Publications; ISBN: 1617294136
[9] The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win; IT Revolution Press; 2013; 1st edition ; ISBN 0988262592
[10]Agile Application Security: Enabling Security in a Continuous Delivery Pipeline; Laura Bell, Michael Brunton-Spall, Rich Smith, Jim Bird; 2017; O'Reilly Media; 1st edition; ISBN: 1491938846