**Image for the blog background (choose one that fit well):**

* <https://pixabay.com/vectors/background-the-background-1086840/>
* <https://pixabay.com/photos/devops-business-process-improvement-3155972/>
* <https://pixabay.com/photos/chain-security-metal-iron-3481377/>

**Title:**

Why and how apply security validations after each web application deployment?

**Abstract:**

Deployment of web application (website, api, etc.) is nowadays an activity that is performed several times a day/week/month by a company. However, how to do you ensure that the content deployed only contains what is expected to be exposed? Let’s explore this question together…

**GitHub repository associated that will be publicly released along with this blog post:**

<https://github.com/ExcelliumSA/PostDeploymentSecurityCheck-Study>

**SEO rules indicated by Mathilde:**

* Paragraphs with fewer than 300 words.
* Keyword used as much as possible: *continuous* *deployment, security, validation*
* Presence of sections.

**Author(s):** Dominique Righetto

**ℹ Note:** In the content below, all figure captions refer to the image file that Mathilde must use when she creates the blog post. So, it's normal if the caption does not describe the figure.

# Introduction

With the rising of the **Continuous Deployment** [1] activity, the frequency at which web applications (website, api, etc.) are deployed have significantly increased. Nowadays it is common to see companies deploying a new version of a web application several times by weeks/months [2].

# Continuous deployment activity has a price to pay

With the increase of the frequency of deployments as well as the full automation of the deployment processes, the risk to introduce a problem allowing to attack a freshly deployed web application significantly increase. To be honest, the validation steps (unit test, integration test, etc.), in a continuous deployment pipeline, are critical because they represent the “watch dog” before the exposure of the application to end users.

A common continuous deployment pipeline is like the following:

Diagram

Description automatically generated

Figure 1: Figure01.png

Often the test steps (Unit, Integration, Acceptance, etc.) are focused on the objective to ensure that the version deployed is functional from a business point of view (features do what they are expected to perform without bug).

It is technically possible to add security focused tests in these test steps to cover the security aspect. Even if it is not the objective of this post to present possible tests, an interesting talk about this topic, by the WE45 company (<https://we45.com/>), is provided here [3].

However, when the deployment on production step is finished, doubts like the following will remain:

* Does the version deployed only expose content that is expected to be accessible by end users?
* Does the production configuration is hardened as expected?

# Doubt removal

To try removing the mentioned doubts, it is possible to add a final validation step to reach this continuous deployment pipeline:

Diagram

Description automatically generated

Figure 2: Figure02.png

This step, triggered automatically once the application is deployed, apply different security focused validations. The objective is to ensure that the application is in a state consistent with a production environment.

If issues are detected, then, two options are possible depending on the issues and level of automation achieved by the company in its continuous deployment activity:

* Option 1: Fix the detected issues leveraging automation on the components affected via web API provided by the components.
* Option 2: Trigger a continuous deployment pipeline to deploy the previous version.

If no issue is detected, then access to end users (or no action) is performed depending on the deployment model of the application [4].

# Technical proposal

The table below provide a list of validations that can be performed in this final post-deployment step. In this table, every tool leveraged was chosen to perform a processing without depending on an online service. The goal is to open the capability to either target an internal (Intranet) or an external (Internet) application. All chosen tools are free and open source.

|  |  |  |
| --- | --- | --- |
| Validation identifier | Validation objective | Tool used |
| VAL00 | Ensure that all HTTP security headers applicable for the application topology are present and correctly defined. | * <https://github.com/ovh/venom> * Venom test plan following the *OWASP Secure Headers Project* recommendations: https://gist.github.com/righettod/f63548ebd96bed82269dcc3dfea27056 |
| VAL01 | Ensure that only a secure protocol is used (HTTPS). | * Curl combined with some bash commands: <https://github.com/curl/curl> |
| VAL02 | Ensure that the TLS configuration is secure according to current standard. | * <https://github.com/drwetter/testssl.sh> * JQ for results handling: <https://github.com/stedolan/jq> |
| VAL03 | Ensure that no sensitive content, secrets, unexpected content is exposed. | * <https://github.com/ffuf/ffuf> * Custom dictionary (text file specific to the application) of items (path/file) that must not be present after the deployment. * Curl commands to verify some potential information disclosure. * JQ for results handling: <https://github.com/stedolan/jq> |
| VAL04 | Ensure that a security.txt [6] file is present to allow security bug reporting in a secure way. | * Curl combined with some bash commands. |
| VAL05 | Ensure that a Web Application Firewall is present in front of the application. | * <https://github.com/stamparm/identYwaf> |
| VAL06 | Ensure that robots.txt [15] file does not disclose any internal application path (absence of *disallow* clause). | * Curl combined with some bash commands. |
| VAL07 | Ensure that directory listing [16] is not enabled. | * Curl combined with some bash commands. |

# Proof of concept

This script [5] demonstrate an example of “low-level” implementation of the validations presented in the previous paragraph. Usage of a shell script allows to strongly customize validations according to the application and its deployment context.

Below is an example of the report generated, providing all the detail about the different validations as well as a final state. The final state can be used to make the pipeline to fail to trigger a rollback or other automated fixation operation:

Text

Description automatically generated

Figure 3: Figure03.png

Overview of the pipeline using GitHub action [7] and the processing time of every step:

A screenshot of a computer

Description automatically generated with medium confidence

Figure 4: Figure04.png

Validation operations stay short in terms of delay, less than 3 minutes. It is important to keep this delay shortest as possible to [8]:

* Not impact parallel deployments of several applications by the continuous deployment platform.
* Provide quick feedback about a deployment, allowing to run a deployment several times in case of need.
* Not monopolize resources for a long-time frame.

# Increase the maintainability

In the previous section, a shell script was used to perform the collection of security validations proposed. Even if it is a direct and effective way to achieve the validation steps, it can become difficult to maintain with the time and the increase of validations steps performed (in addition to being a platform specific script). For the steps requiring only to perform HTTP requests (no execution of local tools like *testssl* for example), it is possible to move the collection of validations to a “recipe” easier to edit, maintain, test and being portable across different operating systems on which a continuous deployment platform is installed.

The tool, named “venom” [9], can help to achieve the migration to a recipe via its “tests plan” approach and its cross-platform support.

This test plan [10] demonstrates how it can be achieved (execution from a Windows machine):

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5: Figure05.png

It is interesting to note that venom can execute local tools [11] and deal with generated report for the assertions part, therefore, it is possible to include operation requiring external tools to achieve a global test plan like the one implemented via a shell script. The only drawback of including execution of external tools is that it broke the portability if the tools are not cross-platform. However, it is possible to keep the portability aspect via the creation of a dedicated ephemeral docker image containing the tools, the venom binary file, and the test plan.

# Go Further

It is possible to add many more security tests, there is no limit. One suggestion can be to ensure that no administration interface, with default credentials, is left accessible, moreover, if the application is based on a product. To achieve this, the tool, named “nuclei” [12] (cross-platform), can be leveraged. Indeed, via its approach based on templates, it provides a collection of “templates” to detect different kinds of administration interface. In case of need, custom templates [13] can be created.

Below is an example of usage of *nuclei* to identify every login panel with default credentials, the tag “default-login” [14] instruct *nuclei* to apply all templates in charge of such detection:

Text

Description automatically generated with low confidence

Figure 6: Figure06.png

After the execution of such command, it is possible to verify if login panels were found by checking the content of the text file generated. If no panel was identified, then the file empty.

# Conclusion

Continuous deployment activity reduces the timeframe between the implementation of a feature and its delivery to end users. It can bring a real advantage from a marketing/sales perspective against competitors. However, it requires to be in full control regarding the product delivered to ensure that new delivered product did not represent a security risk for the provider. This blog post provided technical hints, to achieve this situation of control, to fully benefit from a continuous deployment activity and you can use them to build your own post deployment security validations strategy.

# Authors

* Dominique Righetto

# References

1. <https://www.atlassian.com/continuous-delivery/continuous-deployment>
2. <https://cloud.google.com/blog/products/devops-sre/another-way-to-gauge-your-devops-performance-according-to-dora>
3. <https://www.youtube.com/watch?v=i34Ihbuslgw>
4. <https://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/>
5. <https://github.com/ExcelliumSA/PostDeploymentSecurityCheck-Study/blob/main/validate.sh>
6. <https://excellium-services.com/2021/05/18/security-txt/>
7. <https://github.com/features/actions>
8. <https://www.atlassian.com/continuous-delivery/continuous-integration>
9. <https://github.com/ovh/venom>
10. <https://github.com/ExcelliumSA/PostDeploymentSecurityCheck-Study/blob/main/recipe.yml>
11. <https://github.com/ovh/venom/tree/master/executors/exec>
12. <https://github.com/projectdiscovery/nuclei>
13. <https://nuclei.projectdiscovery.io/templating-guide/>
14. <https://github.com/projectdiscovery/nuclei-templates/tree/master/default-logins>
15. <https://github.com/danielmiessler/RobotsDisallowed>
16. <https://portswigger.net/kb/issues/00600100_directory-listing>