IT 351 Cloud Security and DevSecOps

Lab 5

Fall 2021 – Professor Josh Edgar

Welcome to the Security part of this course, where we really start integrating security automation into our DevSecOps pipelines! In this Lab, we will work hands-on with software composition analysis, specifically by using the OWASP Dependency Check tool, a utility that scans the third-party libraries imported by your code for any known vulnerabilities.

We will add the Dependency Check job to the pipelines you submitted for your Pipeline Project, since the underlying infrastructure that is built thus far doesn't matter to these downstream jobs. We will experiment with running the Dependency Check job without any vulnerabilities, as well as with vulnerabilities while enabling/disabling the "Fail on CVSS" flag that decides whether or not the pipeline will continue running. This is a great first Lab to kick off the latter half of our semester, because you will get to see how security automation helps make risk management decisions about your code in real time!

Before you begin:

- **Your Pipeline Project jobs must all run and complete successfully with green checkmarks.** If any of the jobs are failing with red Xs, you must fix these jobs before starting Lab 5, or you may end up having to scrap your Lab 5 work and start it over. It isn't important if your code installed/deployed all of the items required by your prompt, but it is important that the job completes successfully.
- Your GitLab Runner instance should be started, if you previously stopped it.
- You should have run the "destroy" stage of your Pipeline Project, terminating the instance that was created.

What you'll achieve:

- Add a Dependency Check stage to your GitLab CI/CD file, allowing the OWASP Dependency Check container to look at the code in your repo, preparing a report about any found vulnerabilities.
- You will run the job multiple times, at one point deliberately introducing known vulnerable packages, so that you can see the job fail your pipeline when the vulnerability is found.
- You will also use GitLab artifacts to export the HTML report generated by the Dependency Check job, which you will turn in for your grade via Canvas.

What you'll turn in with this lab:

- The HTML report generated by OWASP Dependency Check, uploaded to Canvas.

Task 1: Update Our GitLab CI/CD File to Add OWASP Dependency Check

By now, you should be pretty used to this process – at some point in our Labs, we always add a stanza to our GitLab CI/CD files to create a new stage and/or job, which adds functionality to our pipeline allowing us to experiment with what we are learning during a given week. In this case, this is a no-frills stanza that simply fetches the OWASP Dependency Check container, and runs the built-in Bash script to scan our repository for vulnerable imports.

1. In VS Code, open your ".gitlab-ci.yml" file.
2. In the top section where our "stages" are listed, add an entry after "apply" and call this new stage "`Dependency Check`" (with quotes).
    a. This is a minor new exercise for us. Did you know that you can come up with custom stage names that have multiple words in them, as long as you surround them with quotes? You can!
    b. Make sure that you are manually typing this, as well as everything else that you add to your code – copy and paste from Word does not work because "these quotes" are different from the quotes used in VS Code.
3. Download the sample GitLab CI/CD code from this assignment's page in Canvas. In between the job code for our "ansible" and "tf-destroy" jobs, add the code for the "owasp-dc" job.

Observe that we have a section for preserving a GitLab "artifact" in this code, similar to how we pass our Terraform plan into our Terraform apply job, and how we passed our Terraform IP address output into our Ansible job as an inventory file.

In this case, though, we are preserving the HTML report that OWASP Dependency Check generates once it analyzes our repo, called "dependency-check-report.html." We add a new line to expire this artifact in one week, rather than immediately, so that we can access it for several days after the job completes.

4. Save your CI/CD file, commit it, and push it up to your GitLab repo.

This will cause your pipeline to execute. In GitLab, under CI/CD > Pipelines, you should see your pipeline running, and the length of job icons should now be one (1) unit longer, accounting for this new OWASP Dependency Check job.

5. Watch as your pipeline runs. You will need to manually kick off your "tf-apply" job, as always. Once your pipeline gets to the "owasp-dc" job, click on it to watch the output. The job takes longer than usual to run – about five (5) minutes to download all of the CVEs from the NVD and check your repo files for them. This runtime is with our relatively small repos, so you should expect this job to take even longer in the real world, where more detailed code would exist in your repo.

6. Because we have not introduced any vulnerable libraries in our code, the job should eventually complete successfully. Because we have asked to preserve our dependency-check-report.html file, we should now see a prompt to the right of the terminal output for downloading the job's artifacts. You can also access this from your Pipelines page, by clicking the download dropdown to the far right side of the pipeline's row.
7. Review the HTML report, and see that there are no identified vulnerabilities.

Task 2: Introduce Vulnerable Libraries

Now that we know how the Dependency Check report looks blank, let's introduce some vulnerable Python libraries on purpose, and see if the job catches them.

8. Download the sample "requirements.txt" code from the assignment page on Canvas.
9. In VS Code, create a new file in your repo's main directory named "requirements.txt" (without quotes). Place the content from the downloaded "requirements.txt" into your "requirements.txt" file. Save your file, commit, and push.

Your pipeline will run again, from the very beginning. You'll still need to manually trigger your "tf-apply" job, even though it will have nothing to do since our Terraform instance is still running (it was never destroyed) and we did not change our Terraform code. Once your pipeline reaches the "owasp-dc" job, watch the job output again.

10. Notice how the "owasp-dc" job still completes successfully. Download the HTML report again and review it. Do you notice any differences?

Task 3: Add a Flag to Fail the Pipeline at a Certain CVSS Score

We should see in our report from Task 2 that there were definitely vulnerable versions of Python libraries found in our requirements.txt code. However, the Dependency Check job still completed successfully, because, of course, it did its job. It can't remediate these vulnerabilities for you, it can only generate the report for your edification. That's not good, because our pipeline could still keep running downstream jobs, deploying this vulnerable code into our cloud environment. Let's use a cool feature of the Dependency Check script to fix that.

11. In VS Code, re-open your ".gitlab-ci.yml" file.
12. Find the "owasp-dc" job code, and look for the line under "script:" where the Bash script is executed. At the end of this line (after `--enableExperimental`), add an additional flag that says "`--failOnCVSS 7`" (without quotes).

Remember in the lecture how we talked about CVSS scores, which determine the severity of a particular vulnerability. Adding the `--failOnCVSS 7` flag tells our OWASP Dependency Check job to send a negative exit code to the job environment if it identifies *any* vulnerability with a CVSS score exceeding 7.

The total average of the CVSS scores doesn't matter – if the Dependency Check job finds ten (10) vulnerabilities, and nine (9) of them all have CVSS scores of 1, but one (1) vulnerability has a CVSS score of 8, the overall job will still fail, even though the average CVSS score of all of those vulnerabilities is only 1.7. The Dependency Check process will still fail because one of our vulnerabilities had a CVSS score of 9 (above 7). We can set that 7 threshold to any number we want, depending on our organization's risk appetite.

13. Save, commit, and push this change to your CI/CD file. Watch your pipeline run again, kick off that "tf-apply" job, and watch the "owasp-dc" job again. What happens now?
14. Download your HTML report from this job. *What do you mean "I can't?"* Oh, that's because GitLab artifacts aren't preserved from failed jobs!

Task 4: Tell GitLab We Want That Artifact Either Way

By default, GitLab does not preserve artifacts, even the ones that you've requested in your CI/CD file, when the job fails. The logic behind this is reasonable: What good is an artifact from a failed job? Don't you only want outputs from successful jobs?

Normally, the answer would be yes, but we're getting a little bit snarky with the way we're developing our pipeline in the name of security. We're *deliberately* making our job fail if our vulnerability scan comes back with bad enough news. GitLab wasn't really designed with this in mind, but there is a workaround – we can declare a specific setting in our CI/CD file that says that our artifacts should *always* be preserved, whether the job fails or succeeds.

15. In VS Code, re-open your ".gitlab-ci.yml" file.
16. Find the "owasp-dc" job code, and in the "artifacts:" section, right above "paths:", add a line that says "`when: always`" (without quotes). Save, commit, and push.

Again, your pipeline runs, you get to that "owasp-dc" job, wait forever for it to conduct its analysis, and the job will fail again. However, this time, you will find that your ability to download the HTML report artifact has been re-enabled. **Download and save this final HTML report, and upload it to Canvas to receive credit for completing this Lab.**