IT 351 Cloud Security and DevSecOps

Lab 6

Fall 2021 – Professor Josh Edgar

Welcome to Lab 6, where we're working with the *other* type of SCA – static code analysis. Interestingly enough, we'll also use this Lab to integrate with the *first* type of SCA we discussed – software composition analysis – because we'll be pulling our OWASP Dependency Check reports into SonarQube.

As discussed in our PowerPoint deck, SonarQube is a SCA (static code analysis) platform that reviews all of the code in our repo, without executing it, and provides feedback as to how we can improve our code to be more correct and secure. It needs its own EC2 to perform the analysis, so we'll set that up in Terraform before working with the analysis job itself. Then, we'll introduce some bad PHP code that insecurely transmits MySQL credentials in plaintext, and see if SonarQube catches it. Finally, we'll connect our OWASP Dependency Check report into SonarQube, so that we can see all of the issues with our code in one place.

Before you begin:

- Your Lab 5 should have been completed successfully (it's OK if the job fails because of that "failOnCVSS" flag, but if your Dependency Check job is failing for any other reason, you must fix that before starting on Lab 6).
- Your "main" branch pipelines in your repo must work. If you have cut separate branches for your labs, you will need to merge back into the "main" branch, because SonarQube only scans main branches for free.
- 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 an EC2 instance to your Terraform code for the SonarQube host, where scans are performed and reports can be viewed;
- Add a Code Analysis stage to your GitLab CI/CD file, allowing a SonarQube Scanner container to take code from your repo and snd it to the aforementioned instance for analysis;
- Introduce some insecure PHP/MySQL code into your repo, and seeing if SonarQube flags it for correction;
- Use a GitLab Artifact to send your OWASP Dependency Check report from Lab 5 into the SonarQube console.

What you'll turn in with this lab:

- The IP address and login credentials to your **SonarQube** EC2 instance.

Task 1: Update Terraform Code to Add SonarQube EC2

Unlike most of the other tools we've used this semester, we can't just run our SonarQube scans in a container on our GitLab Runner. SonarQube requires that a separate EC2 instance be deployed to conduct the scans, as well as provide a GUI to view scan results. We could provision this manually, but that wouldn't be very DevSecOps of us! As such, let's make some changes to our Terraform code to deploy this instance using automation, and pass the IP address off to our downstream Code Analysis job.

1. In VS Code, open your "main.tf" file.
2. Download the sample Terraform code from this assignment's page in Canvas. Place the "resource" stanza for a new SonarQube EC2 instance below your existing "resource" stanza for your original Terraform-created instance. Place the "output" stanza for fetching SonarQube's IP address below the existing "output" stanza for your original EC2 instance.
   Once completed, the order of your code should be:
   `resource, resource, output, output.`
3. Update the Name tag in the new "resource" stanza to reflect your own name for the SonarQube instance.
4. Log in to the AWS Management Console, and navigate to the EC2 Console. On the left sidebar, find the "Security Groups" section.
5. Create a new Security Group for your SonarQube instance – it can be named whatever you like – with an inbound rule allowing access to port 9000 from any IPv4 address.
6. Once the Security Group is created, take note of the Security Group ID it is assigned (starts with "sg-"), and update the `vpc_security_group_ids` line of the SonarQube Terraform resource to utilize this Security Group ID. Make sure to also update the `key_name` attribute to reflect the same private key name that you use in your other Terraform instance.
7. Save your "main.tf" file, but don't commit or push anything yet.

Task 2: Add SonarQube Job to GitLab CI/CD File

Now that we have Terraform set up to create the SonarQube EC2 that will conduct our scans, we can go about the normal steps of creating our GitLab job for invoking the SonarQube scan, which, as always, involves pulling their container image from the Docker Hub. However, you'll notice that there a few other housekeeping steps in here, where we'll have to go back to our Terraform apply job to capture the other output of our SonarQube EC2 IP address, so that we can connect to it from our SonarQube job.

8. In VS Code, open your ".gitlab-ci.yml" file.
9. Add a new stage at the top, called "analyze" (without quotes) *after* your "Dependency Check" job.

10. Download the sample GitLab code from the assignment page on Canvas. (This code only contains the SonarQube job that is being added – all other changes described herein will need to be done manually.) Place the new "sonarqube" job code below your "owasp-dc" job code.
11. Scroll up to your "tf-apply" job code. In "script," below the line where you pipe your original Terraform instance's IP address into the Ansible inventory file, add a new line that will pipe your SonarQube instance's IP address into an environment variable file. Remember, this should all be on one line, even though Word breaks it up:

    ```
    - echo SONARQUBE_IP=$(terraform output --raw sonarqube_ip)
    >> sonarqube.env
    ```

12. In the same job code, under "artifacts," at the same indentation of "paths" but below where the Ansible inventory file is declared for storage, add the following code that will ask GitLab to preserve the sonarqube.env file we just created:

    ```
    reports:
          dotenv: sonarqube.env
    ```

    Notice how we are calling this a "dotenv" report. ".env" files, in GitLab, specifically contain environment variables, similar to the ones we program into GitLab from the Variables web interface. In this case, though, we are trying to capture a variable from our running pipeline, so that we can reuse that variable in a downstream job.

    We also need to go up to our "owasp-dc" job and add one line, indented, right beneath the job title – this will prevent our failing OWASP job from killing our entire pipeline, allowing us to continue playing with SonarQube:

    ```
    allow_failure: true
    ```

13. Save your CI/CD file, commit the changes from both files from Tasks 1 and 2, and push.

Your pipeline will run – and don't forget, it will take quite a while with that OWASP Dependency Check job – however, it will ultimately fail. This is because we have not yet configured our SonarQube "project" for scan results to be captured, and that's because we do not know our login credentials for our SonarQube EC2 instance until it has already been created. As such, we intentionally need to run a bad pipeline to capture those details, which we can then use to set up our SonarQube project, provide those details in our repo, and then run the pipeline again, properly.

Task 3: Set Up SonarQube Project

14. Return to the AWS Management Console, and navigate to your EC2 Instances table.

15. Select your newly-created SonarQube instance, then select Actions > Monitor and Troubleshoot > Get System Log.
16. You will be shown the complete system output of your SonarQube instance being created. The random password, generated when SonarQube was installed by the AMI, is contained in here – protected by your AWS Console log in, so that no one else can locate the default password. Scroll up in the system log until you find something that looks like this:

```
[   58.745123] bitnami[566]: ********************************************************************************
[   58.747336] bitnami[566]: #                                                                              #
[   58.750869] bitnami[566]: #        Setting Bitnami application password to 'AZXOOKyg5ZFh'                #
[   58.752994] bitnami[566]: #        (the default application username is 'admin')                          #
[   58.755746] bitnami[566]: #                                                                              #
[   58.758369] bitnami[566]: ********************************************************************************
```

Save this password somewhere, as you will need to provide it to the professor to receive your grade for the Lab.

17. Locate your SonarQube instance's public IPv4 address – either in the Instances table, or in the outputs mentioned in your Terraform apply job terminal in GitLab. In your browser, open a new tab, and navigate to this IP address, but first adding :9000 to the end of the IP address so that you are connecting on port 9000. For example, if your SonarQube instance's public IPv4 address is 192.168.1.1, then in your browser, you would navigate to http://192.168.1.1:9000/.
    Important: Make sure your URL does not have "https" in it, as SonarQube does not communicate over SSL in this case. If you are following the IP address link from the AWS Management Console, you will need to edit the URL to remove the "s" so that your protocol is only "http."
18. When prompted to log in to SonarQube, use the username "admin" (without quotes) and the password from your system log (without quotes).
19. Once logged into the SonarQube Console, click "Add a project." When prompted, choose "Manually" from the project types. For your project key, use a string with no spaces, such as "your-name-lab6" (without quotes). The display name can have spaces, so feel free to use something like "My Lab 6 Project" (without quotes).
    Click the "Set Up" button to create the project.
20. You will then be prompted to generate a token, sort of like a key pair for SonarQube. You can name this anything you want. I choose to reuse my project key – "your-name-lab6" (without quotes). Click "Generate" and then copy down the token hash once it appears.
21. Click "Continue." Don't worry about the remaining prompts.
22. Download the sample SonarQube code from the assignment's page on Canvas. Create a new file in your VS Code repo named "sonar-project.properties" (without quotes), and place the sample code into this file.
23. Edit the code so that it reflects the project key and description that you created in step 19, and the token that you were provided from step 20. Save the file, commit and push.

Your pipeline will run again, but this time, the SonarQube job should not fail because it will be able to scan your repo and provide results to your project in the SonarQube Console. Allow your pipeline to run fully – expect this to take up to 10 minutes. Don't forget to manually kick off your "tf-apply" job. Now is a good time to take a short break if you'd like one.

24. When you return to your computer, in the SonarQube Console, click the "Overview" tab to return to an overview of your project. Notice that you have 2 bugs being reported by SonarQube already – click that link, and you will find that it is already suggesting code improvements to your HTML file from Lab 4!

Task 4: Add Bad PHP Code to Your Repository

We are already seeing SonarQube making some modest recommendations on how to improve our "index.html" file that is in our repos from Lab 4. (If you don't see that alert because you deleted your HTML file earlier on, don't worry about it.) Now, let's deliberately introduce some really bad PHP code that transmits MySQL credentials in plain text, and see what SonarQube has to say about that.

25. Download the sample PHP code from the assignment's page on Canvas. In VS Code, create a new file in your repo called "bad.php" (without quotes), and place the sample code in there. You do not need to make any changes to this bad PHP code. Simply save it into your repo, commit, and push.
26. Let your entire pipeline run again. After it completes, return to the "Overview" tab in the SonarQube Console. Now, you should see one "Security Hotspot" get flagged, with a red grade of "E" (there are no Fs in SonarQube, strangely). This is a severe issue. Follow the link to observe that SonarQube has found that we are transmitting MySQL passwords in plain text within bad.php, and that this is a high priority for our developers to fix.

Task 5: Import Dependency Check Report into SonarQube

Finally, let's connect one of our upstream job artifacts into SonarQube so that we can see all of our vulnerabilities in one place. SonarQube has a plug-in that supports the parsing of OWASP Dependency Check reports, provided that they are in XML format. Let's install that plug-in, get our "owasp-dc" job to throw the reports into SonarQube, and then review the overview one last time.

27. In the SonarQube Console, navigate to the "Administration" tab at the very top, then go to the "Marketplace" sub-tab on the next page. Click the "I understand the risk" box about installing third-party plug-ins.
28. In the list of plug-ins, scroll down and locate the "Dependency-Check" plug-in, and click "Install."

29. Your SonarQube EC2 instance needs to be restarted for the install to take effect, but we can do that from the SonarQube Console: Click the "Restart Server" button in the alert that has just appeared at the top of the window.
30. In VS Code, open your ".gitlab-ci.yml" file and scroll down to the "owasp-dc" job code.
31. We need to change our Dependency Check report output to be in XML format, as this is the only way SonarQube can read it. To do this, add the following at the end of the current UNIX command under "script:"
    `-f XML`
32. Now that we've changed the format of our report, the name of our artifact to preserve has also changed. Under the "artifacts:" section, change the ".html" extension to ".xml," so that we are now preserving a file named "dependency-check-report.xml".
33. Scroll down to your "sonarqube" job code. At the end of its UNIX command under "script:", append the following so that the scanner can capture the XML file from the GitLab artifacts:
    `-Dsonar.dependencyCheck.reportPath=dependency-check-report.xml`
34. Additionally, we need to ensure that our SonarQube job waits for the OWASP Dependency Check job to finish, since it needs an artifact from it. To do this, in the "needs:" section, add an additional line:
    `- job: owasp-dc`
    This line of code should be at the same indentation as the other needed job (tf-apply).
35. Save, commit and push.

Your pipeline will run one more time.

36. Return to the SonarQube Console. You will need to log in to the Console again after the instance has rebooted. Click the "Projects" tab at the very top to exit the Administration section of the Console, then click on your project's name to get to its overview.

Notice that now, you should have approximately 10 vulnerabilities being flagged by SonarQube, as it has reviewed the disappointing report that OWASP Dependency Check produced as a result of our "requirements.txt" file from Lab 5.

This completes this Lab! Hopefully you have been able to see how powerful SonarQube is as a static code analysis tool, and how it can also import data from software composition analysis. Both types of SCA can be handled from this Console if your DevSecOps pipeline is working properly! This makes it exceptionally convenient for DevSecOps teams to figure out what vulnerabilities need to be urgently addressed, as well as what code in their repos needs improvements.

Please provide your SonarQube instance's IP address/port 9000 URL, as well as the password you captured from your EC2 system log, as a text submission for the assignment on Canvas. The professor will log in to your SonarQube Console to verify that the Lab has been completed.