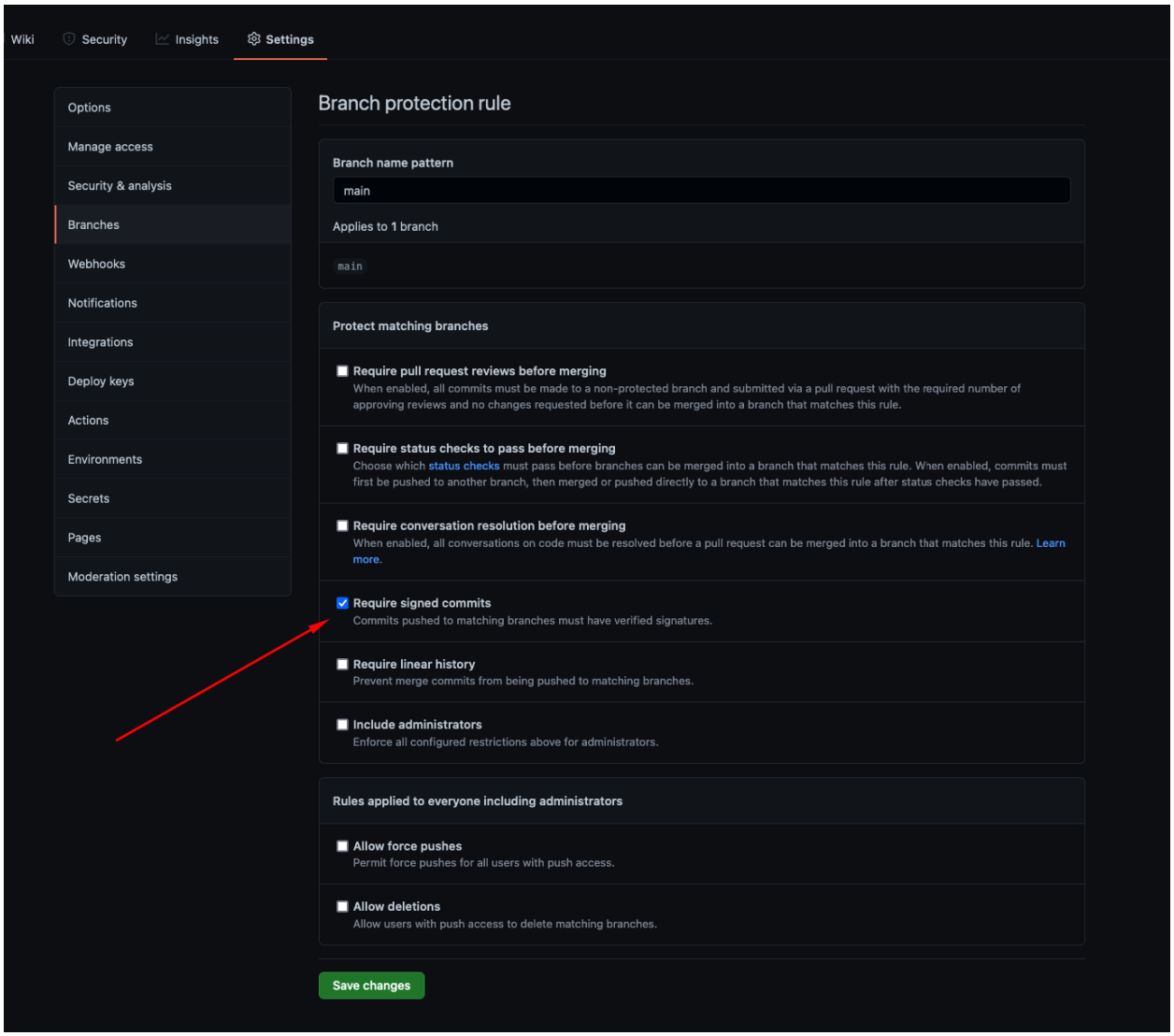
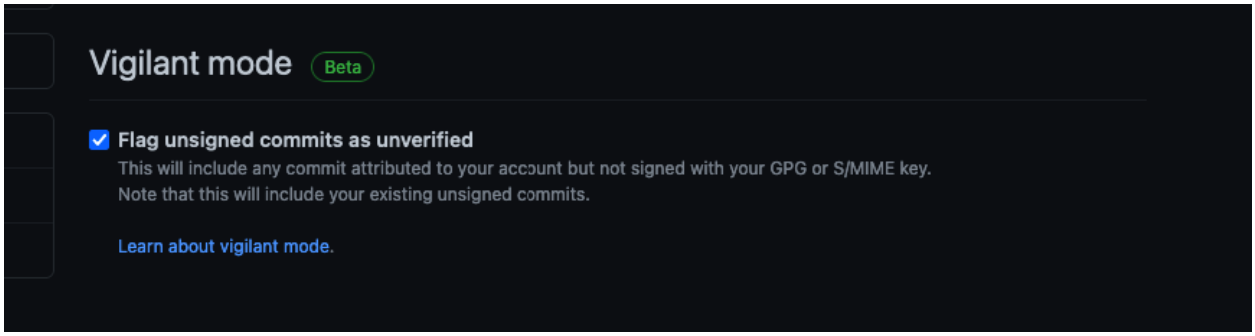
Commit Validation

1. *(We will do this step as a group due to the fact that Branch protection is a paid feature. You can do this yourself if you have a paid GitHub account)*
   Configure protected branch settings to require signing in your newly created repository from the template provided. We recommend your main branch, but it's honestly up to you what branch you select. Be sure to use that branch from that point forward.
   - This will allow you to require that anyone committing to your main branch (or whatever regex pattern you enter) have signed their commits. Note that this doesn't work for repository owners so you will be able to directly commit to main as the owner without signing. See image below:



2. Personal settings configuration:
   Set vigilant mode (under SSH/GPG keys in user settings). See image below:



- This will enable you to see visually in GitHub what commits have been signed and which ones haven't with ease.

3. Next, generate GPG key (instructions on generating a GPG key)
   - Note that you scan skip installing the GPG CLI. It's already on your system.
   - The type of key and password aren't super important for this exercise as we will delete these at the end of the workshop. Please type/enter/pick whatever you'd like, just remember your choices!

4. Add GPG key locally and to GitHub per the instructions you have open from GitHub.
   - You'll need to run the command below in order to get your Linux environment set up to use your GPG key with git. Also be sure to use the username and email from your GPG key in the git config commands.
   export GPG_TTY=$(tty)
   git config --global user.name "FIRST_NAME LAST_NAME"
   git config --global user.email "MY_NAME@example.com"
   git config --global user.signingkey KEYID_FROM_GITHUB_STEPS_FOR_GPG
5. Pull your repository
   git clone https://xyz
6. Change directories into your repo and make a code change (like trying to add a new dependency or a UI change), stage it, and then make a verified commit.
   You will want to generate an "access token" from GitHub to use as your password when git prompts you when you're pushing your code - pick read and write scopes for the sake of the exercise.
   - **Keep this token around! You will likely be prompted to enter it at every push today.**
   git commit -a -S -m "Enter your commit message"
   git push
   - Stop to take a look at how the commit looks in GitHub when you review the commit history.
   - You can review the commit history locally and see the status of any individual commit using the command below.

   git verify-commit <ID>

7. Make an unverified branch and see if it will be allowed to merge into your base branch
   Make a branch
   git checkout -b unsigned-branch
   Make a code change, stage it, and then an unsigned commit.
   git commit -a -m "unsigned"
   git push --set-upstream origin unsigned-branch
   (We will do this step as a group due to the fact that Branch protection is a paid feature. You can do so if you have a paid GitHub account)
   Make a pull request in the GitHub UI to go back into the main branch
   - Check to see if the PR is able to be merged or if your branch protection rules are working and block the pull request from going through.

8. In-toto is a much more involved version of the simple actions taken above (https://github.com/in-toto/demo) and can be customized heavily to achieve a number of effects.

Software Bill of Materials (SBOM) generation

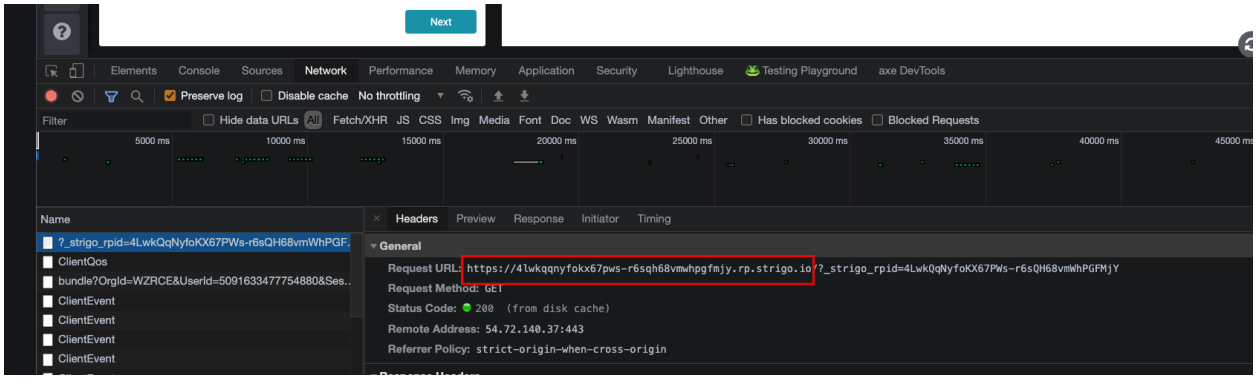
(Cyclone DX SBOM plugin is already in the plugins)

1. **cd spring-petclinic**
   ./mvnw package -DexcludeTestProject=true
   - BOM is in the target directory by default
   - It's worth taking a moment while this runs to explore Cyclone DX and the various things you may find in a SBOM - https://cyclonedx.org/use-cases/
   - Additionally take a read of the US Government's guidance around SBOM to understand where trends may be going due to the recent increase in focus on cybersecurity events - https://www.federalregister.gov/documents/2021/06/02/2021-11592/software-bill-of-materials-elements-and-considerations

Upload to Dependency Track

1. By default the Dependency Track UI will not know where to look for the Server because Strigo dynamically generates the URLs for each app. We need to configure that now.
   First, get the right URL for the Server
   - Open the Server tab in Strigo UI (it may look like "Dependen...". Hover them and see which one is the server)
   - Right-click, open inspect/developer tools
   - Get the URL (minus query parameters and trailing slash) from the network tab (click the refresh wheel in Strigo if not present) See image below, URL will look something like "https://messyalphanumeric-string.rp.strigo.io".
   - Configure the "API_BASE_URL" with the URL you just copied in the dependency track

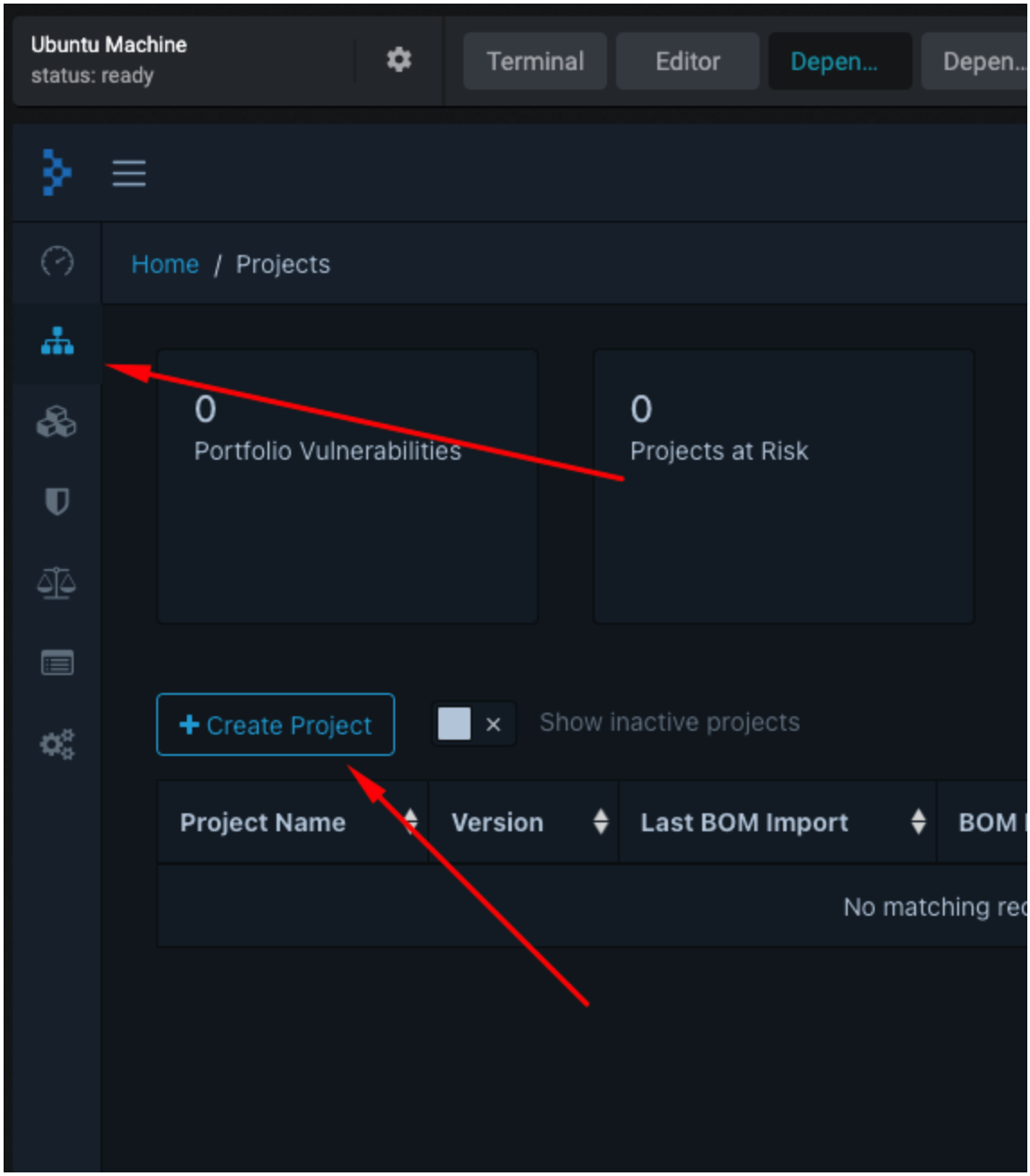
settings in your terminal, following the prompts below



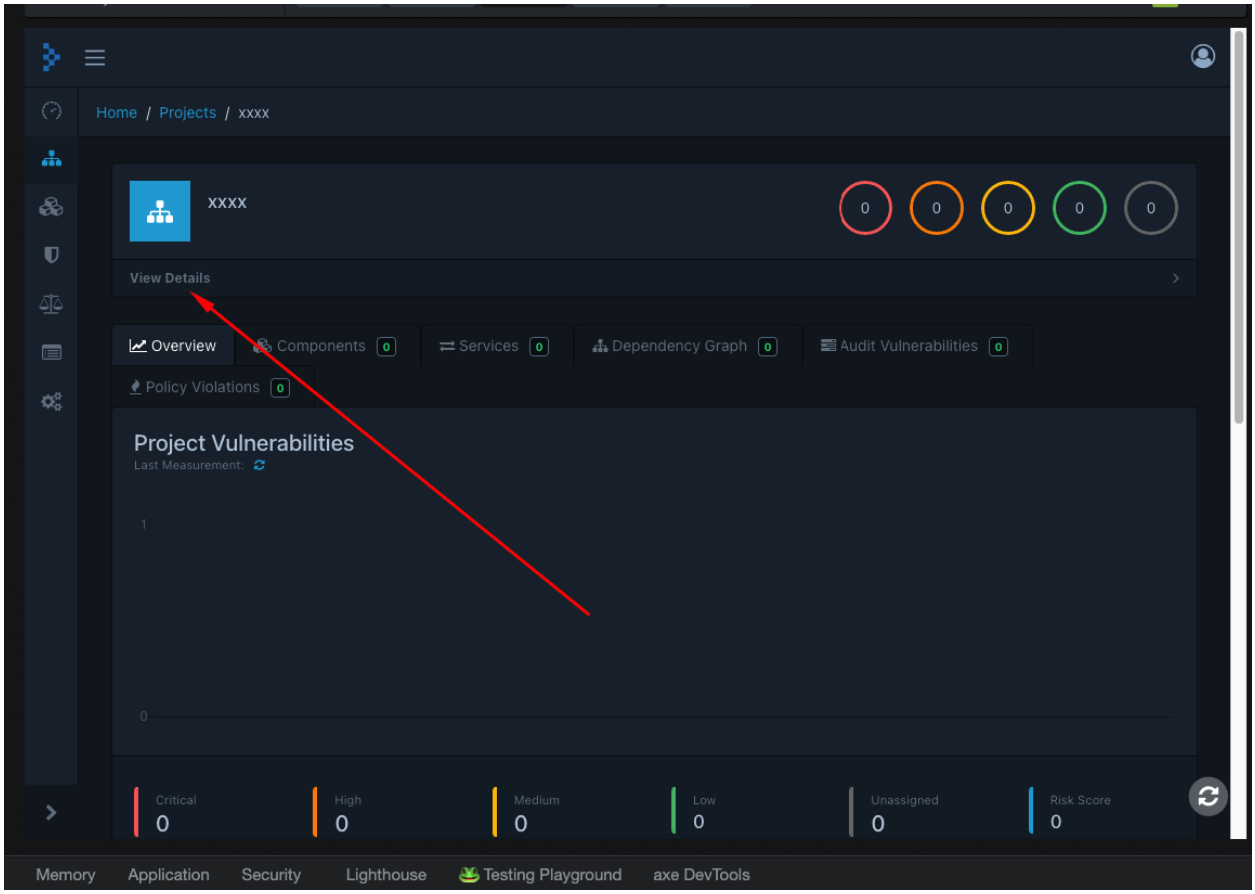
sudo vi /dependency-track/docker-compose.yml
Replace the base url in the UI application
Restart the UI docker container.
cd /dependency-track
docker-compose up -d

2. Go to the Dependency Track UI, click the refresh icon in the lower right of Strigo to get the new settings, and then login using the information below.
   - Username: admin/Password: admin (It will ask you to reset)
   - Please be patient! The server can be slow to respond.

3. Post BOM
   - Make a project in the Projects UI for Dependency Track UI. The metadata doesn't matter. See image below:
   (Note: The UI may tell you that the creation of the project failed, but please refresh before trusting it!)



- Grab the object ID/project ID for your API call from the UI as shown below by clicking View Details. See image below:



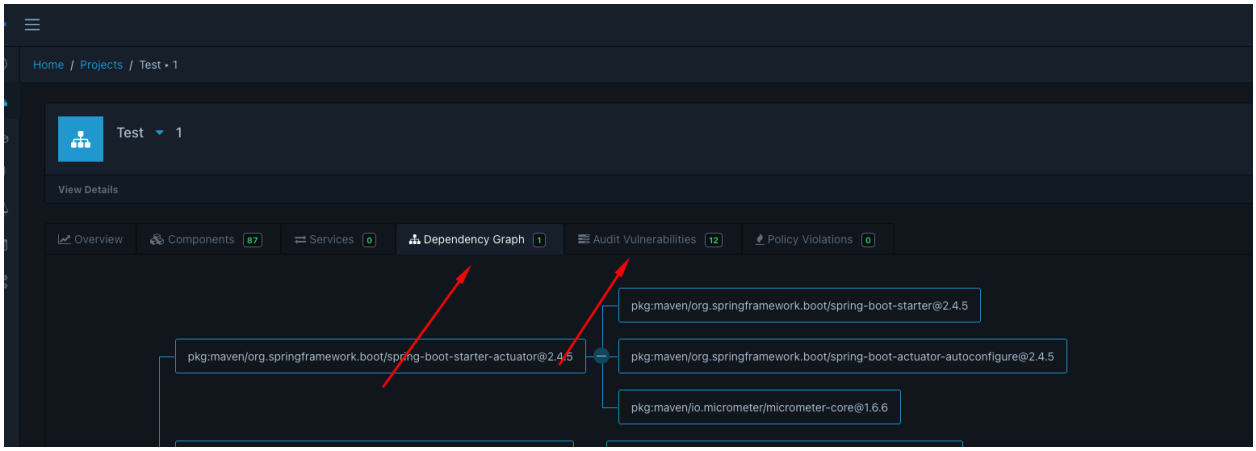
- In the target dir of your compiled application, find bom.xml and post it back to the dependency track server (using the URL you retrieved earlier and the path to bom.xml in the "bom" parameter)
(Note: that there is no need for API tokens as the server has been pre-configured to disable authentication for ease during the workshop)

You will know you're successful when you see a "token" response.

cd ~
cd YOUR_GIT_PROJECT/spring-petclinic/target
curl -H "Content-Type: multipart/form-data" -F "project=YOUR_ID" -F "bom=@bom.xml" -X "POST" "URL/api/v1/bom"

4. Review the uploaded BOM in the Dependency Track UI under the project you just created. See image below:
- How many CVEs are there?

- Is there anything odd in the dependency graph you might not have expected?



It's not trivial or foolproof, but here's ***one*** way to do it.

We are going to generate our final image with the Tanzu Build Service (TBS), which effectively extends the open source project kpack. It has multiple benefits in that in bundles all of the various layers of a build into a single interface.

At each step feel free to dig a little deeper and see if you can learn a bit more about the underlying images and resources.

1. Setup your kubeconfig which is used under the hood by TBS (and kpack)
   mkdir ~/.kube
   cp /config ~/.kube/config
2. Look at the base images available. Note that a "stack" is comprised of both the base image and the image on which the container will be built. They can be different images, but don't have to be.

   kp clusterstack list
   - Could come from anywhere (like Tanzu Application Catalog - TAC). You can read more about "stacks" here and get a better understanding of what those are.
   - To see the specific "stacks" you are seeing here and their description within TBS you can follow this link.

3. Look at the buildpacks available. These buildpacks show what are the available language runtimes to be used during build time. The "store" is equivalent to a repository of sorts, within it there are a number of buildpacks - hence what you see in the status of the default store. Read more on "stores" here.

   kp clusterstore list
   kp clusterstore status default

4. Look at the combined builders (base images/stack + buildpacks/store)
   kp clusterbuilder list
   - This is the reconciliation loop around the base image, buildpack, and source code that really is the magic here. Updating an underlying base image (stack) and/or runtime environment (store/buildpack) can be automated to push an update to all running applications in a cloud native environment with only a singe change. This is because the clusterbuilder creates the loop looking to keep all of its underlying components up to data - hence if source code changes, then a new build needs to happen. Similarly if the underlying base image (stack) changes then a new build needs to happen. Those builds can be automatically watched for and pushed to all running environments!

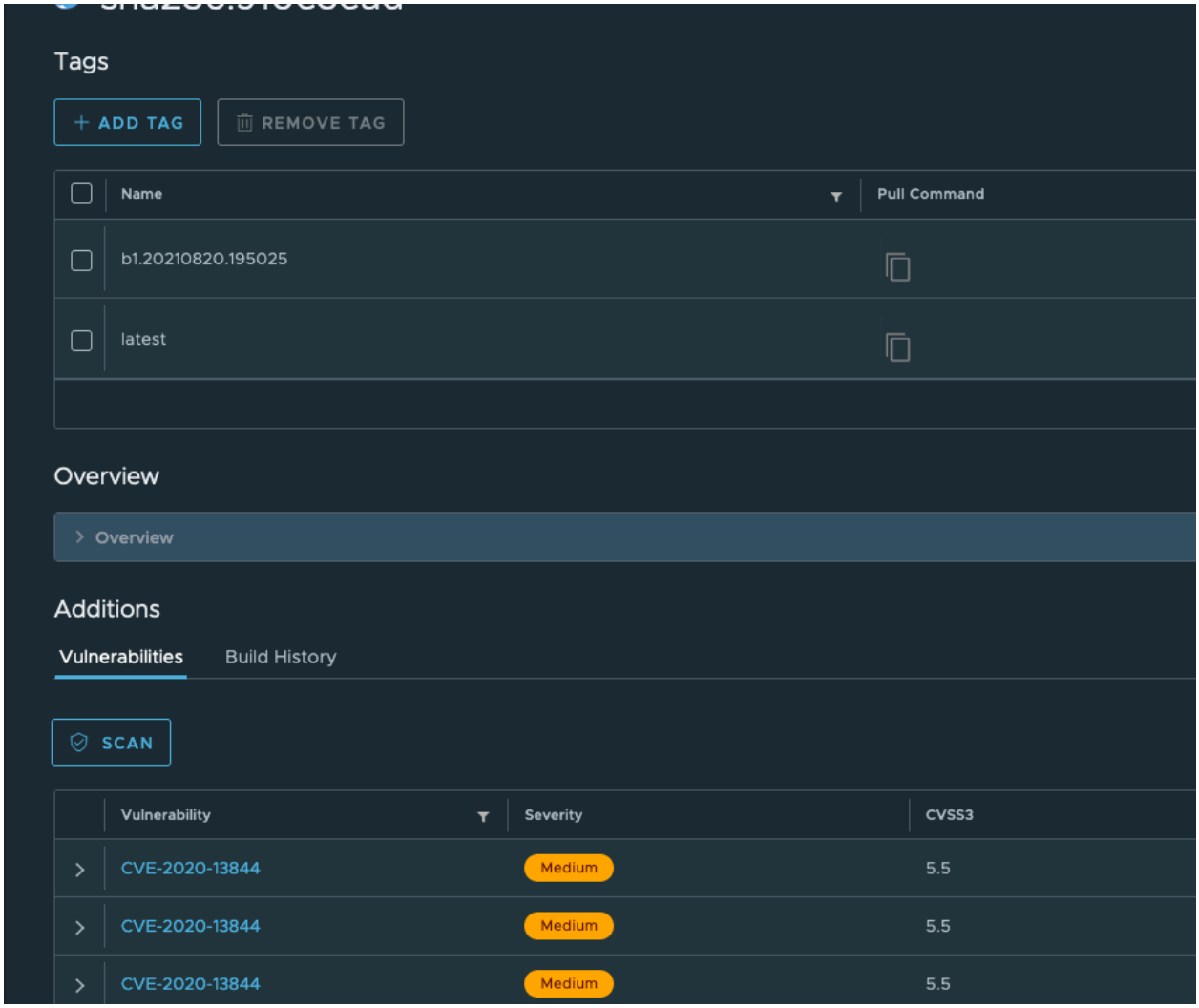
5. Build your own image and have it pushed to Harbor. Note the output for an interesting view into what's happening. **Use lowercase letters and throw something unique like your initials into the image and artifact names so we don't have collisions.**

   kp image create YOUR-UNIQUE-IMAGE-NAME --tag harbor.proto.tsfrt.net/supply-chain/YOUR-UNIQUE-ARTIFACT-NAME --git YOUR-HTTPS-GIT-URL --cluster-builder base --sub-path spring-petclinic --wait

6. Check out the Software Bill of Materials (SBOM) built in TBS and it's general build output.
   - Note that the output of the plain status command shows what were the resulting pieces used within TBS to generate the image (i.e. what buildpacks, what base image, etc.) So a different view of contents than we saw in our other SBOM.
   - Compare the bom command's output against the output from your previous cyclone-dx SBOM generation in exercise 3 (cat /PATH-TO-APP/spring-petclinic/target/bom.json). Note the differences - particularly notice that they are different formats and potentially even slightly different contents.

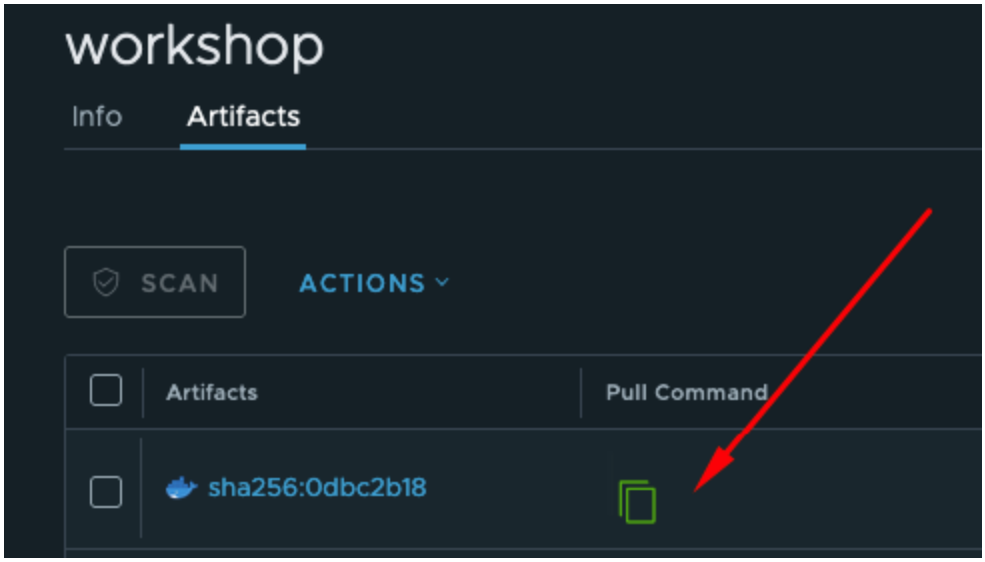
   kp build status YOUR-UNIQUE-IMAGE-NAME
   kp build status YOUR-UNIQUE-IMAGE-NAME --bom | jq .

7. Go to harbor and see the built image. It should be under /supply-chain/YOUR-UNIQUE-ARTIFACT-NAME.
   - Look at the Trivy Scan outputs for your image in Harbor. See image below:



- Compare with the results you saw in Dependency Track. Any differences? (An answer: Trivy has MORE to scan than the dependency track scan did - think underlying operating system, linux dependencies, etc.)

1. Pull the container out of Harbor.
   For this you will want to get the pull command from harbor for your image push. See image below:



docker pull ARGS-COME-FROM-HARBOR

2. Verify who built it
   docker images
   docker trust inspect [IMAGE_ID]
   Oh look, no signature!
   - TBS has a signing mechanism we didn't turn on, but things like https://github.com/sigstore/cosign are other options worth exploring. Take a moment to review cosign, look at how it manages certificates and the ability to validate artifacts (the GIF on their demo is a good preview).
3. Run your image with docker locally. A successful output is a big long string of characters!

   docker run -d -p 8089:8080 [IMAGE_ID]
4. View your application by clicking the tab for Spring Pet Clinic