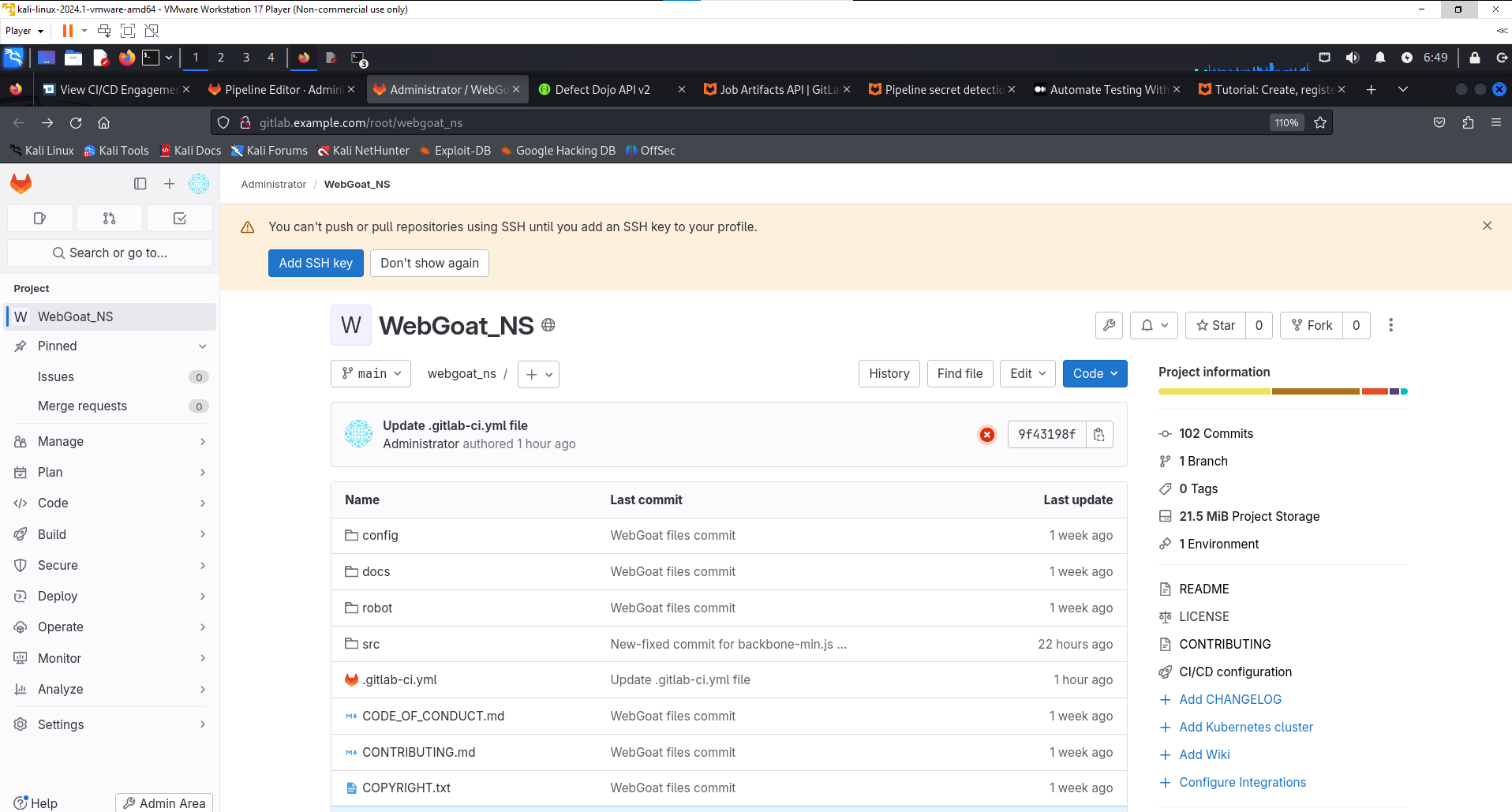
**Project 3 - CI/CD security scanning using GitLab pipeline**

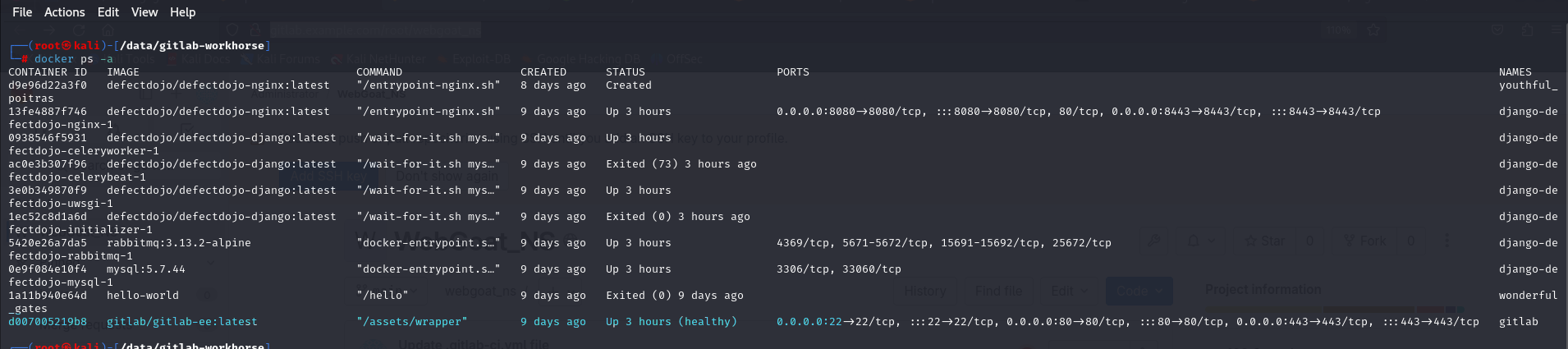
**Aviv Nataf, Omer Bartfeld, Hadar bar asher**

In this project, we were asked to set up a GitLab environment and deploy the WebGoat application to some new project, then, we were asked to run multiple security scans such as:

* Secret detection
* SCA
* SAST
* IaC
* Container scanning
* DAST

We will introduce the pipeline creation and then will show each scan separately, the vulnerabilities each one causes, and how to fix these issues.

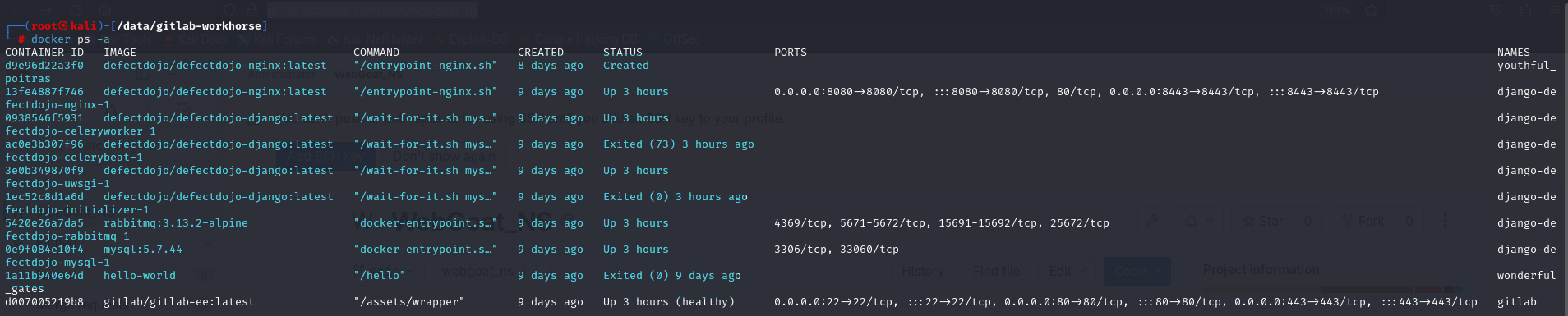
1. **GitLab environment setup:**  
   The process of making the pipeline is the same one that we learned in the class, we first of all pull the GitLab’s docker image and run it as a container, after that, we define our username and password to check into the local server that we construct, to make the process of creating a new blank project and then commit all the files of the WebGoat project at the end.  
   The project at the final step, before all the other steps we took, looks like this:   
     
   

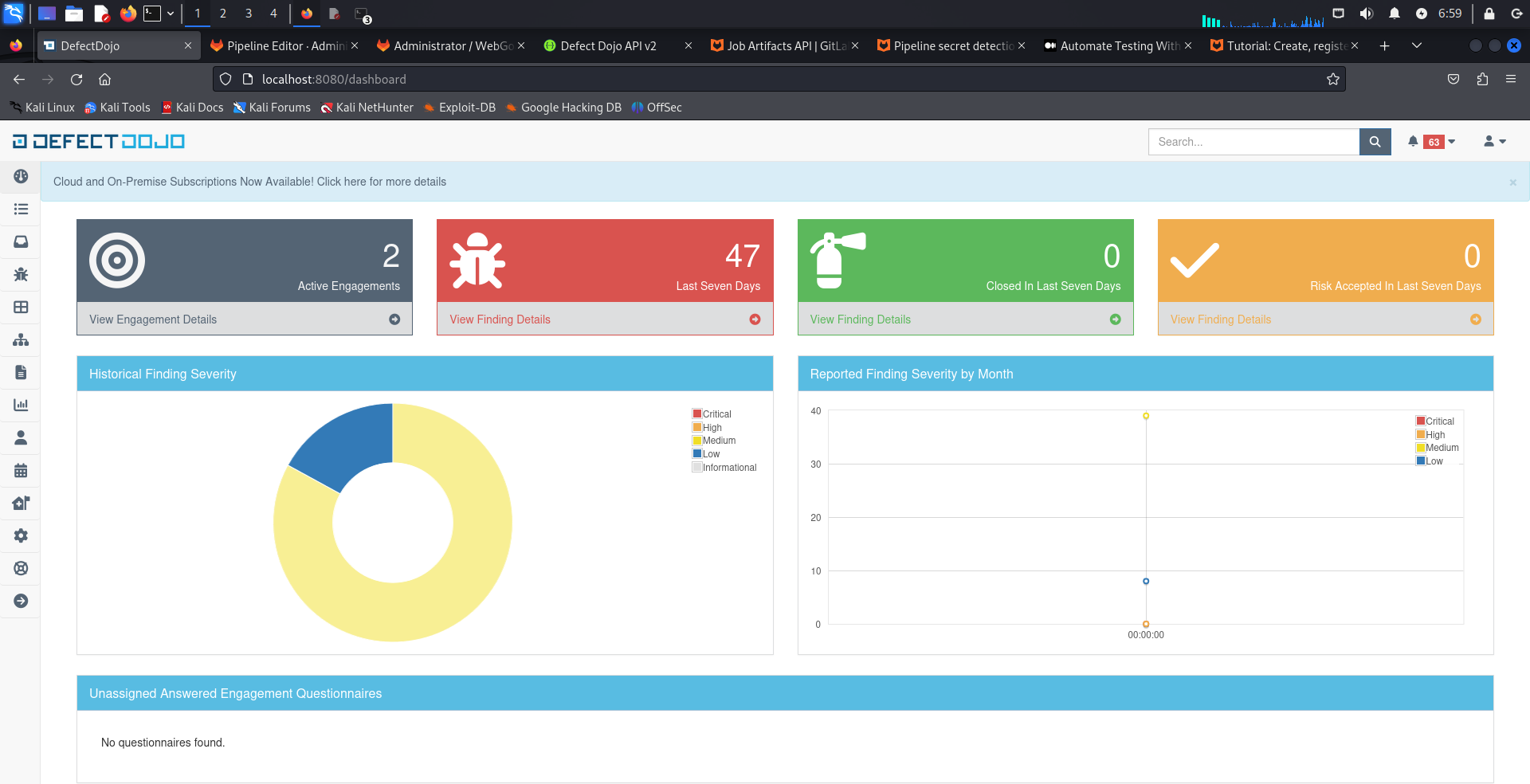
As you can see, the URL is [**http://gitlab.example.com/**](http://gitlab.example.com/)**,** which proves our work on setting up all the environments from scratch.  
The docker container associated with this server runs (marked in blue):   


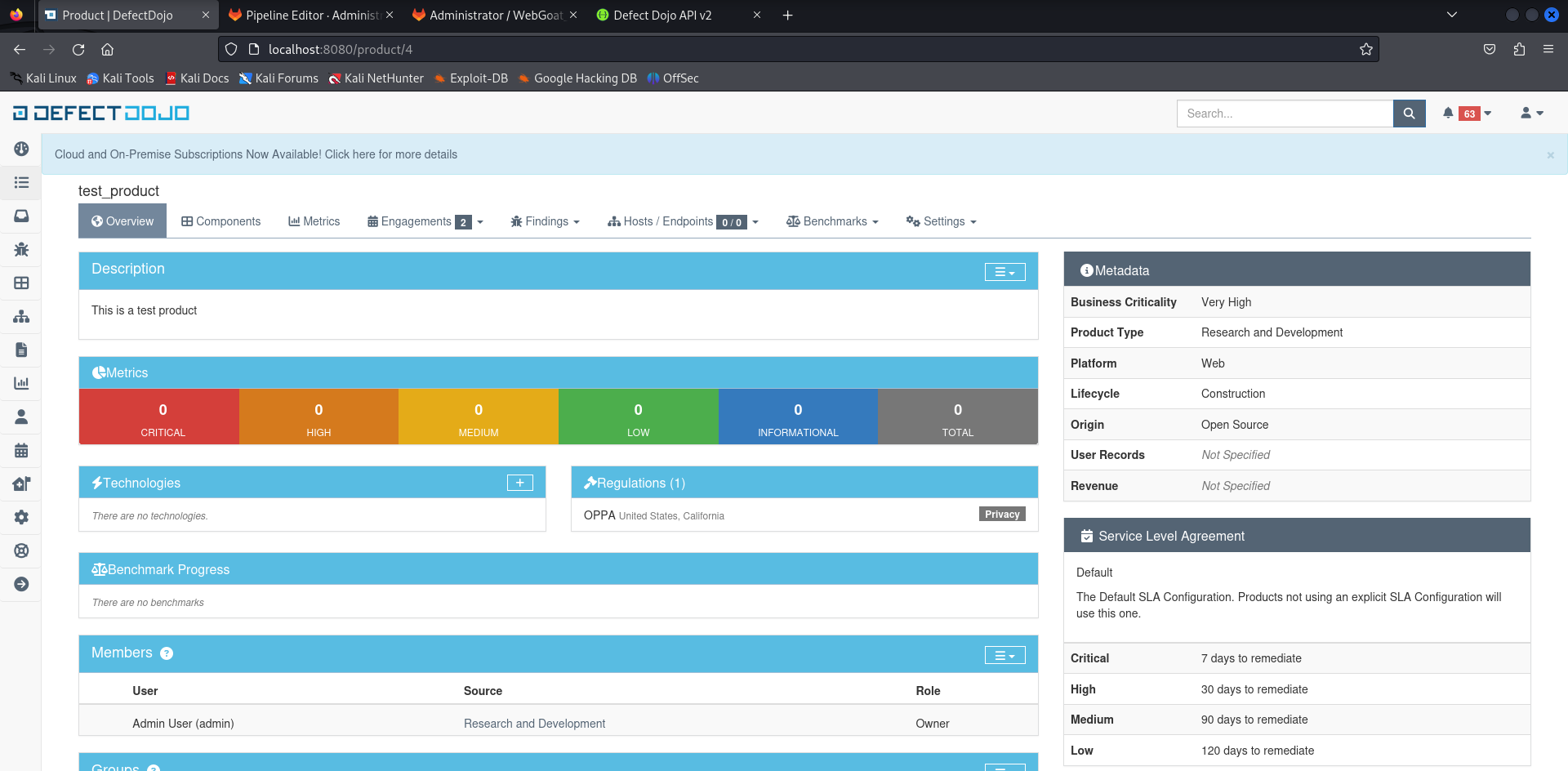
1. **DefetDojo SetUp:**

For the second setup step, we define all the DefectDojo setup, created products, engagements, and tests.

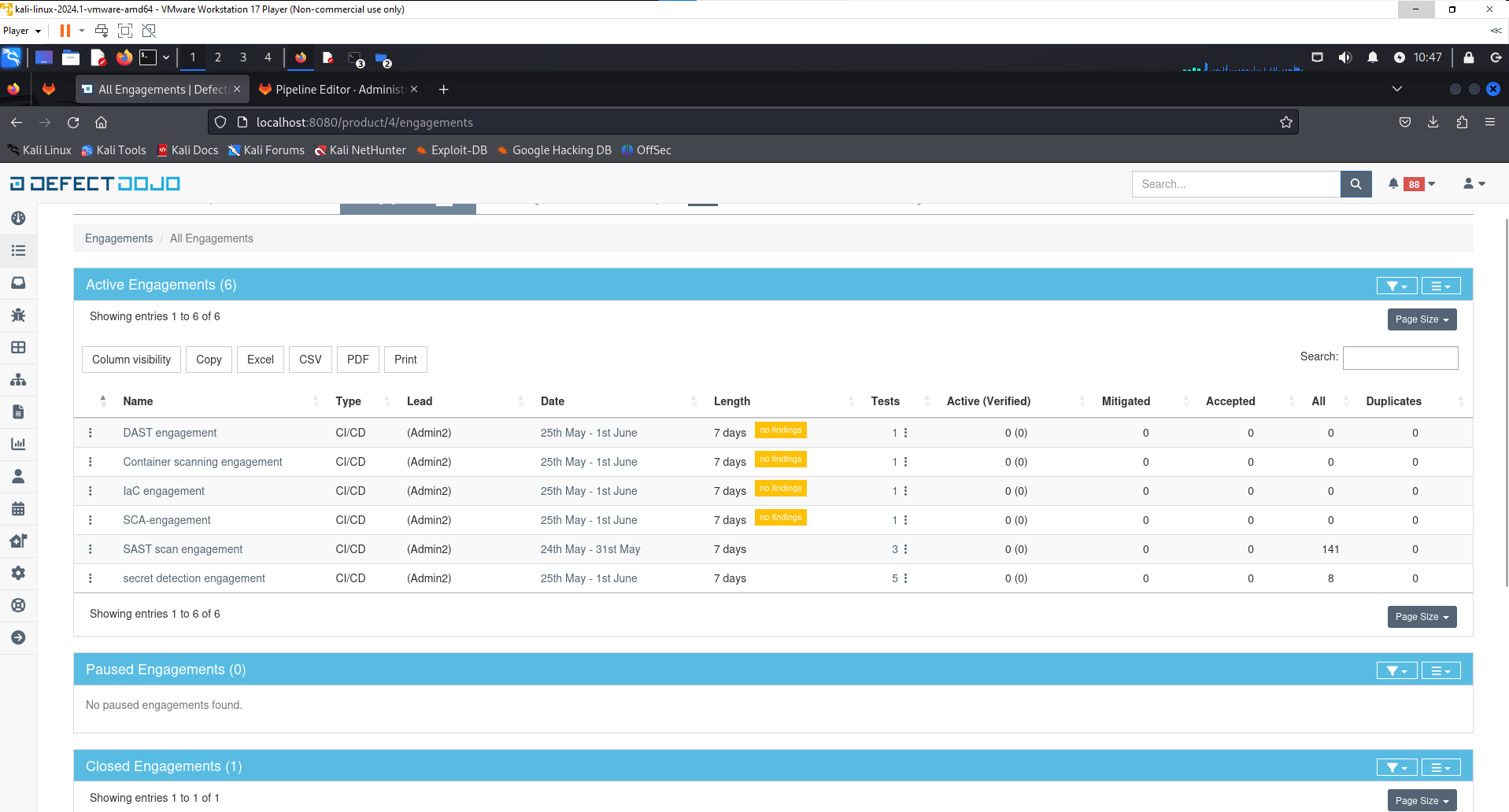
We pulled Defectdojo’s image from their GitHub repo and made some modifications in our VM to make this server up, the final result of DefectDojo’s server on port 8080 is the following (marked in blue):



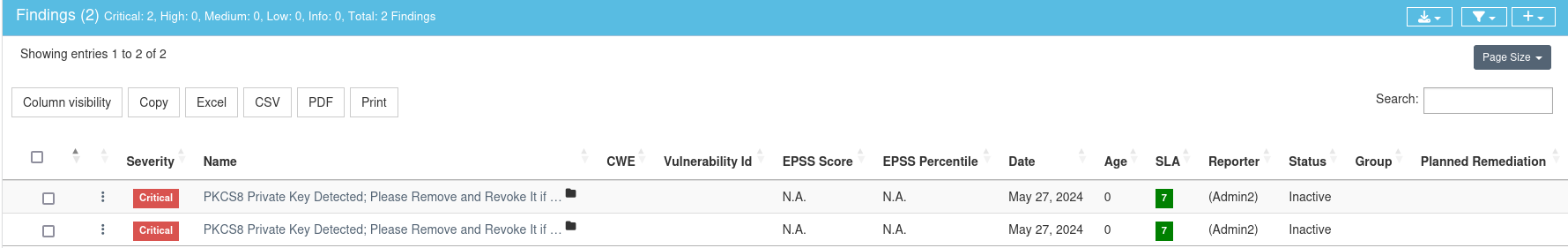


As you can see, the URL for our defect dojo's server is [**http://localhost:8080/dashboard**](http://localhost:8080/dashboard)**,** which proves the fact that **the server runs locally** and on **our VM**.   
  
  
  
  
  
In addition, we created the following product that will be associated with our pipeline in the next steps:   


This test product will store all the information about each scan type and will provide extra information about each one and how to handle it.  
The next step is to link an engagement that will store multiple tests for each scan and scan:

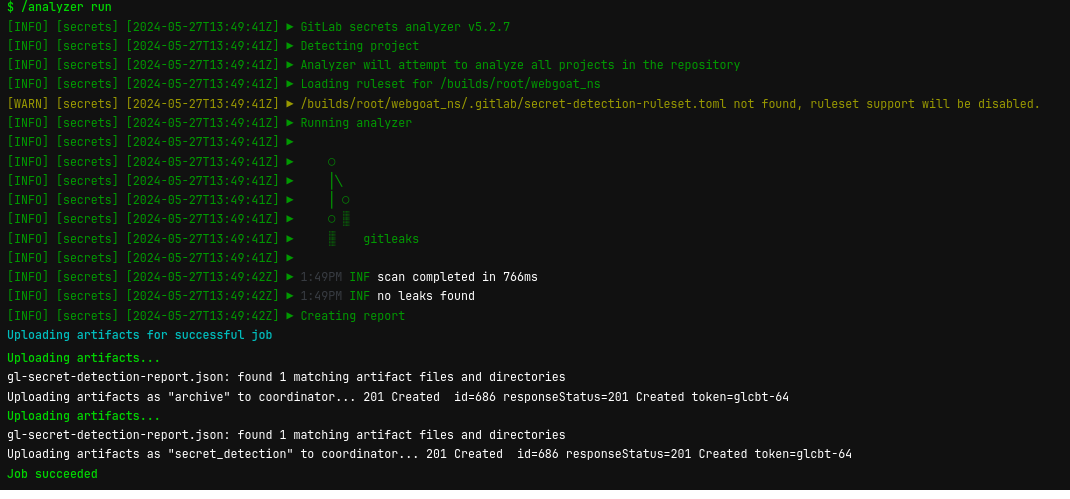


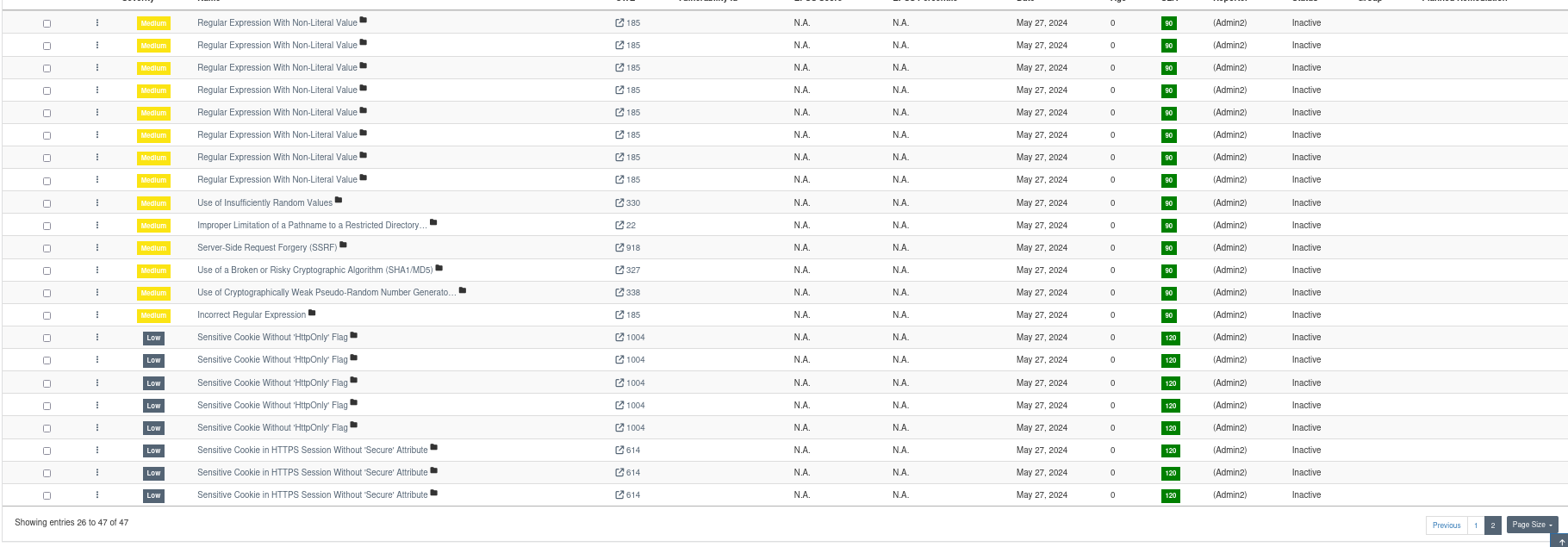
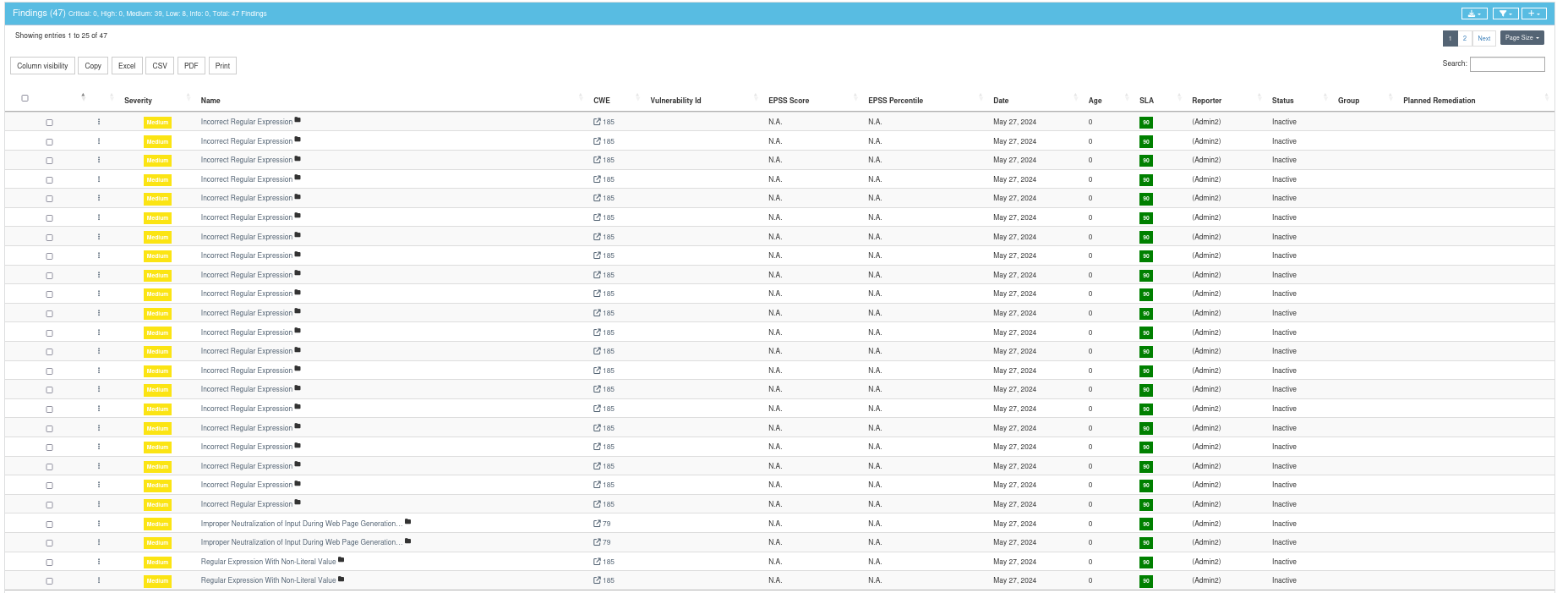
For each test we used to run, we created an engagement that was associated with it, to show the results separately and in some kind of order.

1. **Pipeline creation process:**For the pipeline creation process, including all the features and tools, we took the default pipeline the GitLab provides, and constructed our new one, **we will insert the target pipeline to the project directory with some comments that explain the pipeline construction step by step.**  
   For here and now, we will provide a high-level explanation of the pipeline:  
   For each kind of scan option that we had to run, we created a new stage inside the test stage itself, to make sure that the reports would be stored in the artifacts to extract the specific file to send to the defect dojo's server to show the results and vulnerabilities in a visualized way.  
   The pipeline includes 3types of stages: build, test, and report:  
     
   In the **build stage**,the project has to be compiled to see some compilation issues (if there are some) and prepare for the next stage.   
     
   In the **test stage**, all the tests that we provide in the **include** section are executed, and a JSON file with all the results will be stored in the project’s directory while we add the ‘artifact’ section and define the ‘paths’ inside.  
     
   In the **report stage**, all the POST requests including all the reporting files that have been generated from the test stage, are passed to the defectdojo’s server to analyze and show the result straightforwardly and conveniently, to let the developers see their expert and improve their secure code abilities.  
     
   In our case, we ignored the **deployment stage,** to be more efficient and focused only on the scanning phase and fixing the vulnerabilities (we don't have production env).
2. **Vulnerabilities that occurred during reports:**  
   In this section, we want to show all the results that cause security issues and how we fix them properly, to make the project highly secure:  
     
   - **Secret Detection Report:**    
   This scan is generally used to identify sensitive information (API keys, passwords, crypto-keys, etc.) hard-coded or inadvertently exposed in the source code. In this secret detection scan, we found the following vulnerabilities:  
   

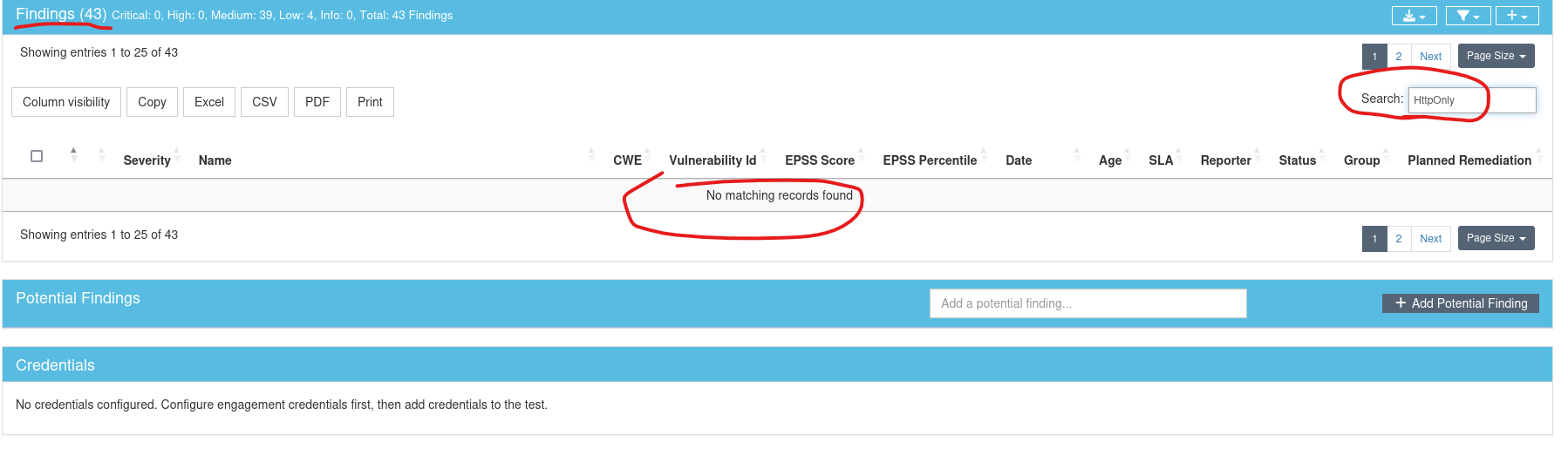
which means that there is a public cryptographic key out there, more particularly, in the ‘src/main/java/org/owasp/webgoat/lessons/cryptography/CryptoUtil.java ‘ file, so to make sure that huge vulnerability will be fixed, we will make sure that all the private and critical information will be securely hidden from the public, the things we can do to make that happened are :

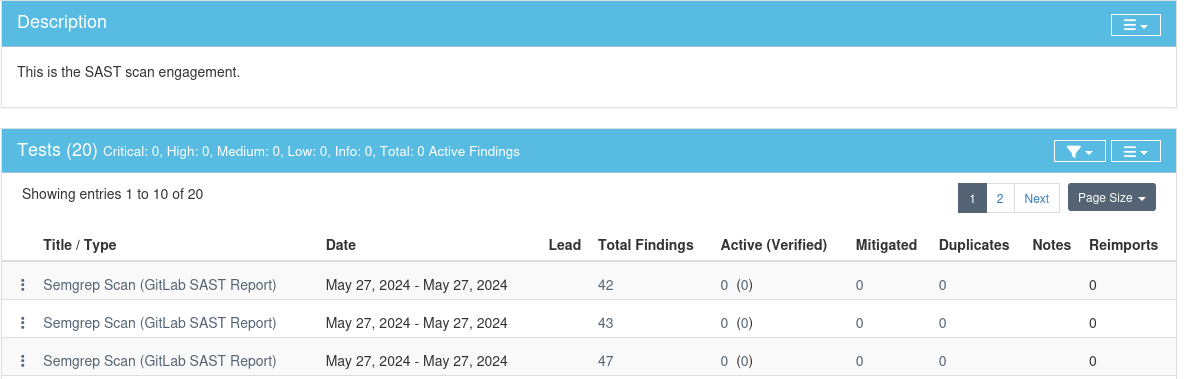
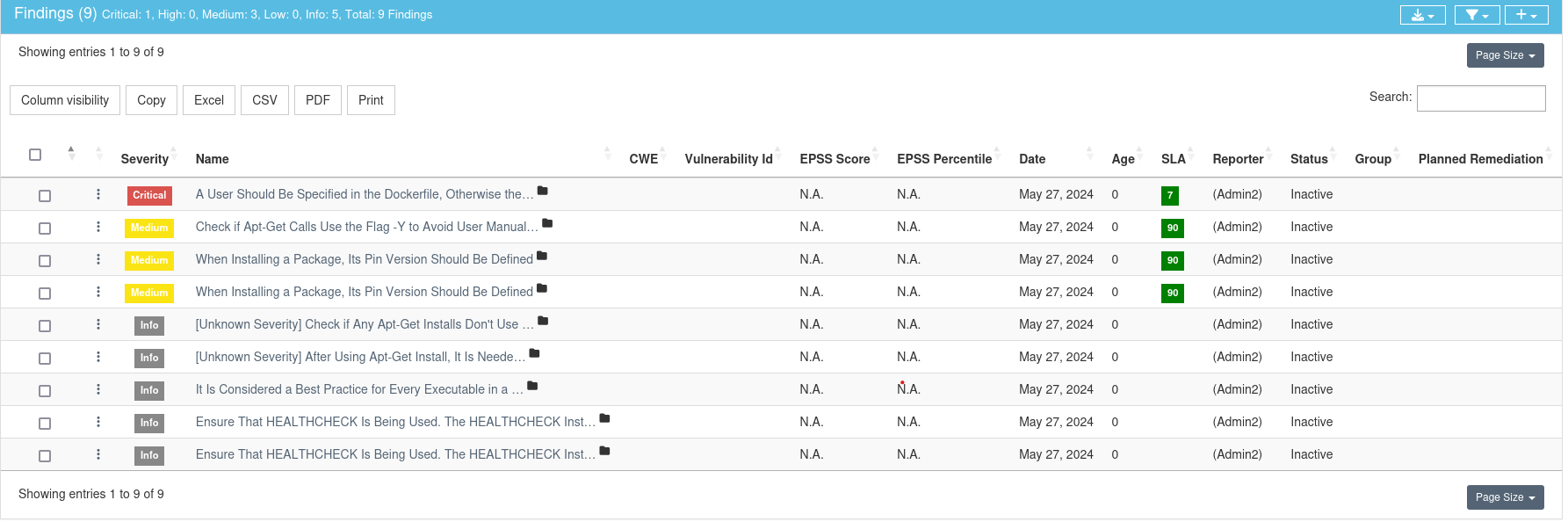
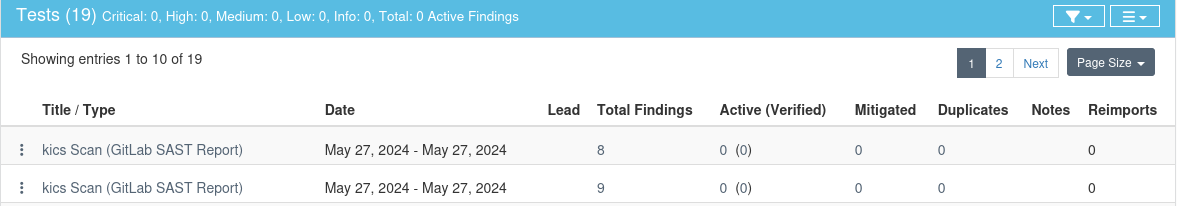
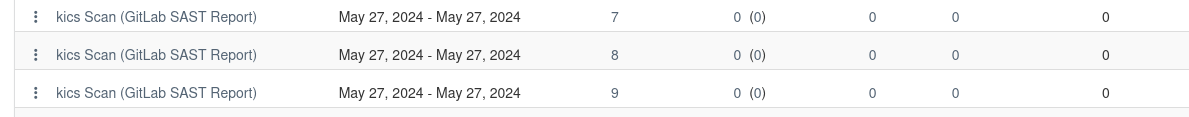
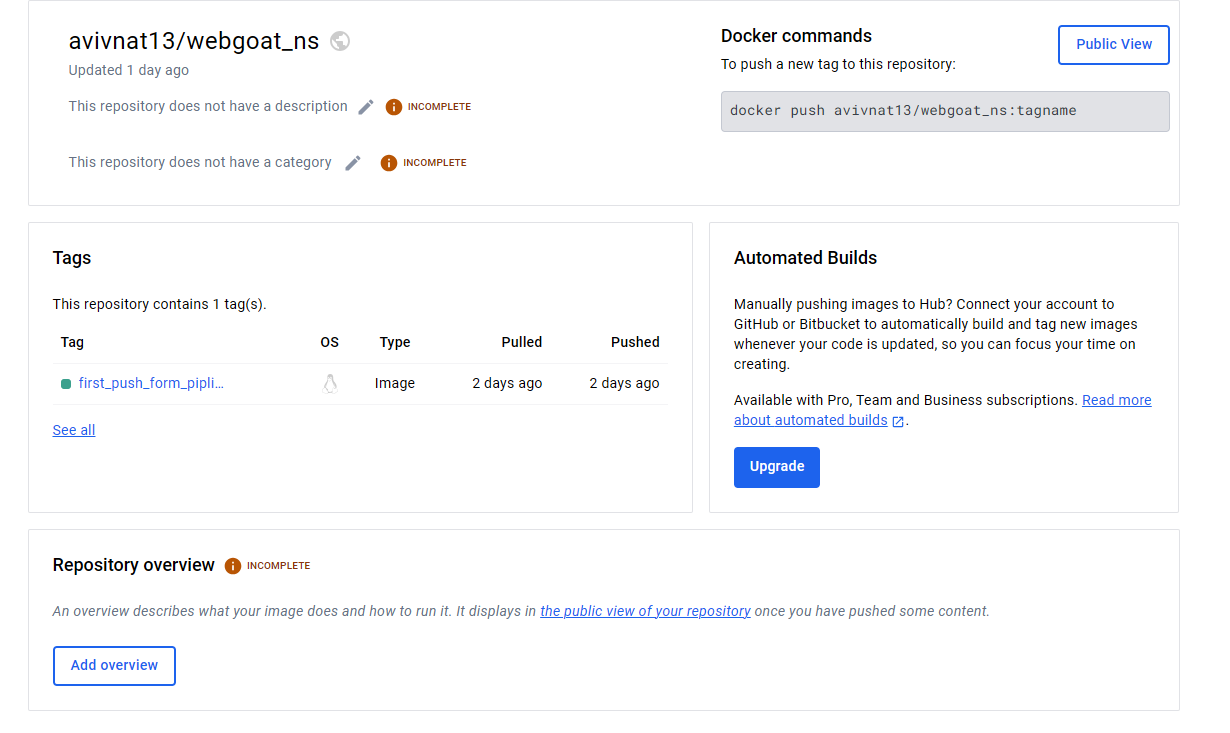
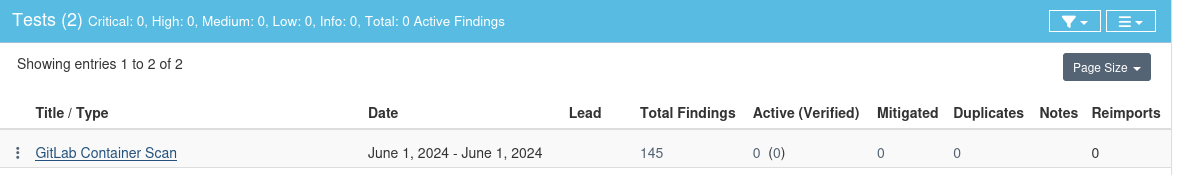
1. Stored the critical variables in a ‘.env’ file.
2. Use the ‘gitignore’ file to avoid these files being committed.
3. Stored all the critical files in the local machine and created functionality that dynamically load all of them while running the project.

In our case, we deleted the problematic file from the project’s repository, (which is ‘CryptoUtil.java’) and then committed again, and the following results arrived:   
  
  


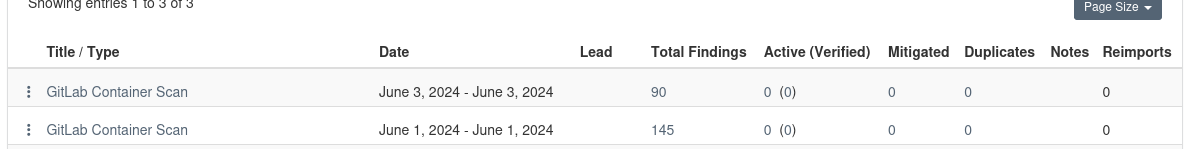
* **Semgrep Scan (GitLab SAST Report):**  In general, a **SAST scan** is used to identify security vulnerabilities in the source code of an application, without compiling or executing the code. It analyzed the codebase for known patterns that could indicate security flaws, such as SQL injections, XSS, buffer overflows, etc.  
  In our case, the SAST scan catches a variety of vulnerabilities and patterns:   
    
  

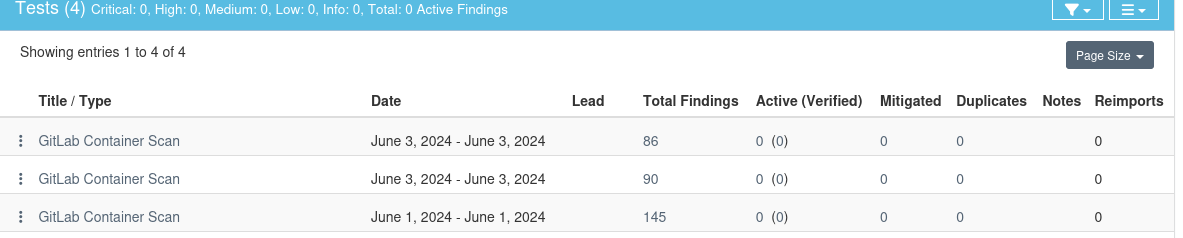
The following vulnerabilities that we fixed are:   
 - **Sensitive Cookie Without 'HttpOnly' Flag**:This issue can cause **XSS from the client side JS script**, there are **unhandled HttpOnly cookies** on the server side, and an attacker that has control on the client side, can take advantage and take the cookies with him to execute cross-site scripting attack, with all the client information.

So in this case, we set the HttpOnly flag to true in the file, and commit the changes:  
  
  
We can see that the t**otal findings reduced to 43** (instead of a total of 47), and when we tried to search for the specific vulnerability, the **HttpOnly** was enough to find that, we **could not find it.**

* **Use of a Broken or Risky Cryptographic Algorithm (SHA1/MD5):**  this vulnerability is a risky one, which means that the application is making use of both SHA1 and MD5 hash functions, while when hashing some stuff, these hash functions can lead to duplicate hashing values. If an attacker finds this vulnerability associated with passwords, he can find multiple accounts’ passwords and hack into them.  
  **The solution that fixes this vulnerability, is to use different hashing functions, such as Argon2id or PBKDF2**.  
  So, in our case, we make use of these hashing functions instead of the old and vulnerable, ones and committed the changes:  
    
    
     
  As we can see, the progress is counting and the vulnerability's finding reduced as well!
* **IaC (Infrastructure as Code) scan:** This scan is generally used to analyze the code to define and manage infrastructure resources (cloud environments, network configurations, etc.) to identify vulnerabilities, compliance issues, and best practices violations.   
  The scan gave the following results:  
    
    
    
  So, we handled the following **two** vulnerabilities:   
   - **A User Should Be Specified in the Dockerfile, Otherwise the Image Will Run as Root:** This vulnerability means that the Dockerfile does not contain any 'USER' instruction, which leads that the DockerFile will run the container as a Root, this can lead to security leaks.  
  To avoid that from happening, **we changed the ‘Dockerfile\_desktop’** that caused the issue, **added another USER that is not a root**, and committed the changes:   
    
    
  By the changes that we made, we prevent this **CRITICAL** vulnerability.  
    
   - **Check if Apt-Get Calls Use the Flag -Y to Avoid User Manual Input:** This vulnerability can let the user insert manual input while running apt-get instructions.  
  So, we fixed this vulnerability again, in the ‘DockerFile\_desktop’ and committed the changes:   
    
  And as we can see, the changes reduced the findings to 7!
* **Container scanning:** This scan is used to identify security vulnerabilities and compliance issues in container images that are used to deploy the application itself. It analyzes container images for known vulnerabilities (OS vulnerabilities), libraries, and application dependencies that are included inside the image.  
    
  To make this scan available, we made the following steps:  
   **1) Created a profile in the docker hub, to create some deploy images:**   
     
  **2) Build the image in the pipeline’s build stage.  
  3) Pushed the source code into the container to make the container scan available, and the following results found inside the defectdojo’s engagement:  
    
  **

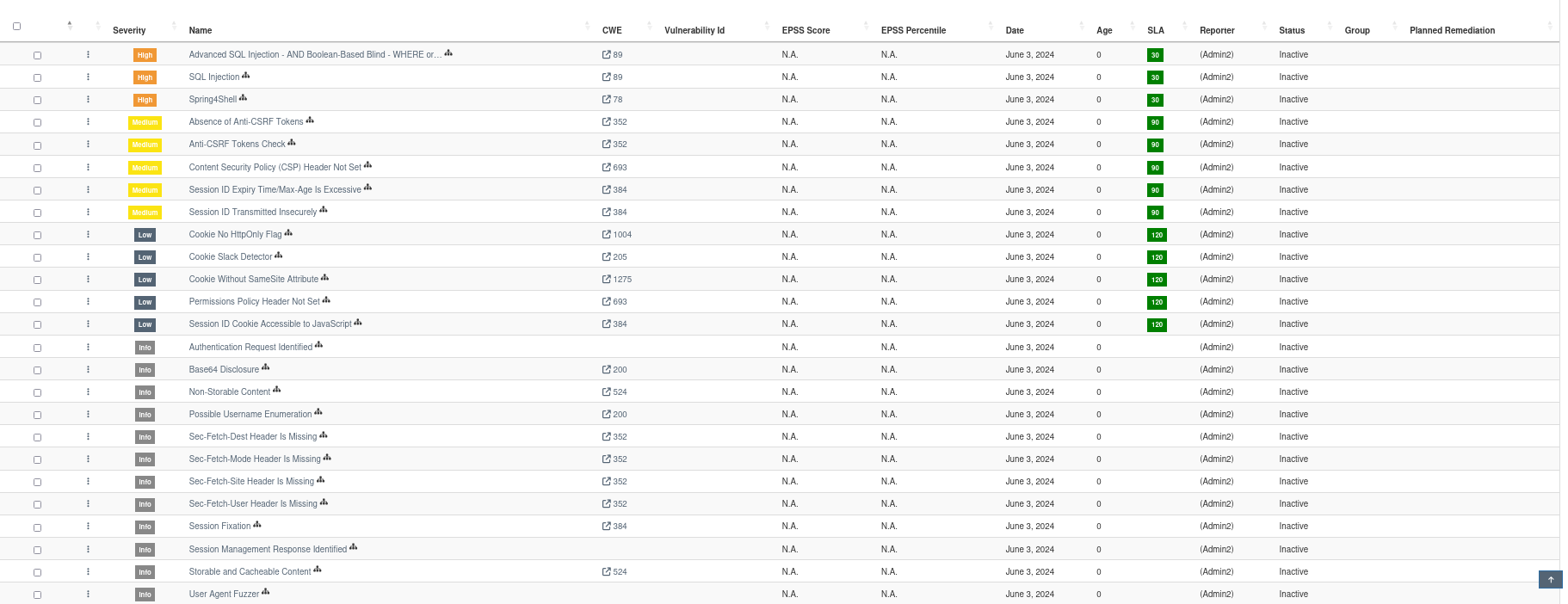
So, in this case, we fixed the following two vulnerabilities:   
 - **CVE-2023-6129 in openssl-3.0.2-0ubuntu1.12:** The POLY1305 MAC (message authentication code) implementation contains a bug that might corrupt the internal state of applications running on PowerPC CPU-based platforms if the CPU provides vector instructions.  
If an attacker can influence whether the POLY1305 MAC algorithm is used, the application state might be corrupted with various application-dependent consequences.  
  
To overcome this vulnerability, we update the image’s version to the latest in the Dokcerfile to make sure we are on the latest version of the base image (which is eclipse-temurin in this case), instead of the default version, which is 21.0.1\_12-jre.  
After this change that we made, we got a huge reduction in the findings section:

  
  
 -**CVE-2022-48065 in binutils-2.38-4ubuntu2.5:** GNU Binutils before 2.40 was discovered to contain a memory leak vulnerability var the function find\_abstract\_instance in dwarf2.c.   
  
To overcome this vulnerability, we must make sure that we have the updated packages that override this vulnerability, so in the Dockerfile, we add another line that **updates packages and installs necessary dependencies**.

After this change had been made, we got the following results after the commit:  
  
  
  
Successfully reduced another 4 vulnerabilities in the findings!  
  
- **DAST scanning (Dynamic Application Security Testing):** This scan is used to identify security vulnerabilities in web applications by dynamically testing in a running state. It simulates attacks against the application to identify common vulnerabilities such as **SQL injections**, **broken authentication**, or **insecure object reference (IDOR).**

The **IDOR** is a vulnerability when attackers can access or modify objects by manipulating identifiers used in a web application’s URL or parameters.

In our case, we automated this scan using **owasp zap (full scan)**, and in the first commit, we got the following findings:



And we will concentrate and solve these two:  
 **- Advanced SQL Injection - AND Boolean-Based Blind - WHERE or HAVING Clause:** in the register file, we noticed that there is a possible SQL injection, While the client is inserting its input, the server does not make the right checks and avoid the possibility of some SQL injection in this case.   
  
In general, SQL injections are a very common vulnerability in web applications, they arise when the user tries to get some data from the web server, that is stored in the database, especially SQL DB.  
With some techniques and manipulations, and the “right” implementation on the server side, the user can get a lot of information from the DB of himself and of others too, and even to delete registries.  
So, to overcome this vulnerability, we need to consider multiple steps:  
 1) Use packages like regex to check every user’s input that has to be passed to the DB.  
 2) Avoid directly passing the user’s input to the DB.

* **Spring4Shell:** the application appears to be vulnerable to CVE-2022-22965 (otherwise known as Spring4Shell) - **remote code execution (RCE)** via data binding.  
  In general, **RCE** is a type of cyberattack where an attacker can execute arbitrary code on a remote system.  
  In our case, **Spring4Shell** from the Spring framework, has this critical vulnerability that allows an attacker the run payloads and gain access to the services of the web application, and access to some sensitive data.   
    
  To overcome this vulnerability, we need to consider multiple steps:  
   1) We have to update the Spring framework that includes patches for this vulnerability.

2) Check and upgrade the dependencies, and make sure they are on the latest and most secure version.

* **Software Composition Analysis (SCA):** Used to identify and manage open-source vulnerabilities within software applications. It involves analyzing dependencies to detect vulnerabilities in third-party libraries or packages used by the application.  
    
  In this case, we used the **Owasp Dependency check scan** to detect and find some vulnerabilities (the full report is in the report’s directory).  
    
  **Unfortunately**, I got some issues in the VM that runs the project (it crashed), and I have no time to fix it before the date, so instead of uploading this report to defect dojo, here are some 5 vulnerabilities and the solutions for each one:   
    
  1) **bootstrap.min.js:** The bootstrap.min.js version 3.1.1 has several vulnerabilities, which might include Cross-Site Scripting (XSS) and other security issues due to the outdated version.  
    
  To overcome this vulnerability, we need to update to the latest version of Bootstrap.  
    
  2) **jquery-1.10.2.min.js:** This jquery-1.10.2.min.js version also includes XSS vulnerabilities.  
    
  To overcome this vulnerability, we need to update the package to its new version too.  
    
  3) **commons-fileupload-1.3.1.jar:** This specific commos-file-upload package has vulnerabilities that could allow for arbitrary code execution or denial of service (DoS) attacks, it's a critical vulnerability that has to be corrected as fast as possible.  
    
  To overcome this vulnerability, we need to upload to the latest version of Apache Commons FileUpload.   
    
  4) **jackson-databind-2.9.9.jar:** This package has an **RCE** (remote code execution) vulnerability, which allows for an attacker to send specially crafted data with hidden harmful commands, and then the library executes those commands in that data, run malicious code on the server and takes control over all the system!  
    
  To overcome this vulnerability, we need to update to at least version 2.10.0 or later.  
    
  5) **library-log4j-1.2.17.jar:** This log4j package also has an RCE vulnerability, that allows an attacker to take control of the server and the whole system, by sending a specially crafted request to the server, which Log4j might process incorrectly.  
    
  To overcome this vulnerability, we need to change the version number to **at least 2.17.1**.