Table of Contents

1. Set of Adequate Interactions

Overall, it was assumed that the files which were provided by the development team produce the desired containers with necessary functionality. Therefore, the primary objective was to retain the original functionality while reducing the attack surface of the built images, enforcing restrictive security policies and creating a secure runtime environment which would result in a secure application. The tables below list the test-case interactions which were used to verify the functionality and security of each application.

App	***Interaction***	***Description/Reasoning***
Database	Connect with the web server.	Part of necessary functionality.
	Retrieve and store user input from the web server.	Part of necessary functionality.
	Run with the provided configuration files from the development team.	Part of necessary functionality.
	Run the container from a stripped image.	Reduced attack surface, limiting the possible actions performed by a threat actor on a compromised container.
	Running with an enforcing SELinux policy while keeping full functionality.	SELinux policies restrict all objects' abilities, be it processes or files, i.e. creating TCP sockets.
	Running with the minimal amount of capabilities and syscalls while keeping full functionality.	Minimal syscalls and capabilities limit the actions that potential attackers can deploy on a container.
	Running with a read-only filesystem while keeping full functionality.	A read-only file system ensures that attackers cannot run malicious scripts.
	Adding persistent volume so database data remains between containers.	Persistent volumes keep a user's data between containers.
Website	A user can access the website through localhost:8080.	Part of necessary functionality.
	Connect to the database to retrieve the stored data and display it to a connected user.	Part of necessary functionality.
	Take user input from the web interface and store it in the database.	Part of necessary functionality.
	Run the container from a stripped image.	Reduced attack surface, limiting the possible actions performed by a threat actor on a compromised container.
	Running with an enforcing SELinux policy while keeping full functionality.	SELinux policies restrict all objects' abilities, be it processes or files, i.e. creating TCP sockets..

	Running with the minimal amount of capabilities and syscalls while keeping full functionality.	Minimal syscalls and capabilities limit the actions that potential attackers can deploy on a container.
	Running with a read-only filesystem while keeping full functionality.	A read-only file system ensures that attackers cannot run malicious scripts.

Table 1.1.: Set of Adequate Interactions

2. Image Hardening & Image Generation

2.1. Dockerfiles & Image Stripping - U2066664

Image stripping was used as the primary approach for generating hardened docker images for each of the applications. A script “strip-image”, created by Mark van Holsteijn and extended by Peter Norris was used to strip the images originally produced by the Dockerfiles to the bare minimum, removing any unnecessary files and functionality. The provided Dockerfile of the database server was slightly altered to ensure that the database is functional after image stripping was completed. Because the used script rebuilds a docker image from scratch, the entrypoint which is used in the original image is not transferred as it is not specifically defined. Therefore the location of the entrypoint script and the original parameters used by mariadb are specifically defined in the altered Dockerfile. Furthermore, the strip-image script flags on lines 265 and 266 were changed from “Z” to “z”. This flag specifies that “the bind mount content is shared among multiple containers” (Docker, 2023a), thus ensuring that the stripped image has the same SELinux labels as the original unstripped image.

The “strip-cmd” script, runs the “strip-image” script with arguments which specify the image which needs to be stripped, the files and packages which need to be included in the new image, the original Dockerfile and the name(s) of the new, stripped docker images to be created. The approach which was used to determine the needed files for the stripped image involved a two-step process. First the necessary executable files were included. This produced errors when run by the entrypoint script. After reviewing the error logs other dependencies can be identified and added to the “strip-cmd” parameter list. This effectively allows us to identify and include only the needed files. Finally another script was run, that would iterate through each line, remove it and attempt to re-strip the image and check whether it runs or not. Further allowing to verify that all the included files are necessary to produce a functioning container. The specific files which were included for each of the images are present in the “strip-cmd” scripts in their respective directories.

2.2. Image Testing Using Snyk - U2066664

The stripped images of the web server and the database were tested using the Snyk docker image vulnerability scanner. Figures 2.2.1 and 2.2.2. below, show the test output for both of the stripped images.

```
[csc@localhost iss-cw4]$ ./snyk-linux container test iss2023/bravo-4-web-stripped_i

Testing iss2023/bravo-4-web-stripped_i...

Organization:      bytute
Package manager:   linux
Project name:      docker-image|iss2023/bravo-4-web-stripped_i
Docker image:      iss2023/bravo-4-web-stripped_i
Platform:          linux/amd64
Licenses:          enabled

✔ Tested iss2023/bravo-4-web-stripped_i for known issues, no vulnerable paths found.

Note that we do not currently have vulnerability data for your image.
```

Figure 2.2.1.: Vulnerability scanner output for the stripped web server image.

```
[csc@localhost iss-cw4]$ ./snyk-linux container test iss2023/bravo-4-db-stripped_i

Testing iss2023/bravo-4-db-stripped_i...

Organization:      bytute
Package manager:   linux
Project name:      docker-image|iss2023/bravo-4-db-stripped_i
Docker image:      iss2023/bravo-4-db-stripped_i
Platform:          linux/amd64
Licenses:          enabled

✔ Tested iss2023/bravo-4-db-stripped_i for known issues, no vulnerable paths found.

Note that we do not currently have vulnerability data for your image.

-------------------------------------------------------

Testing iss2023/bravo-4-db-stripped_i...

✗ Medium severity vulnerability found in golang.org/x/sys/unix
  Description: Incorrect Privilege Assignment
  Info: https://security.snyk.io/vuln/SNYK-GOLANG-GOLANGORGXSYSUNIX-3310442
  Introduced through: golang.org/x/sys/unix@v0.0.0-20210817142637-7d9622a276b7
  From: golang.org/x/sys/unix@v0.0.0-20210817142637-7d9622a276b7
  Fixed in: 0.1.0


Organization:      bytute
Package manager:   gomodules
Target file:       /usr/local/bin/gosu
Project name:      github.com/tianon/gosu
Docker image:      iss2023/bravo-4-db-stripped_i
Licenses:          enabled

Tested 3 dependencies for known issues, found 1 issue.

Snyk wasn't able to auto detect the base image, use `--file` option to get base image remediation advice.
Example: $ snyk container test iss2023/bravo-4-db-stripped_i --file=path/to/Dockerfile

Snyk found some vulnerabilities in your image applications (Snyk searches for these vulnerabilities by default). See https://snyk.co/app-vulns for more information.

To remove these messages in the future, please run `snyk config set disableSuggestions=true`
```

Figure 2.2.2.: Vulnerability scanner output for the stripped database image.

The scanner did not find any vulnerable paths in either of the images, however it detected a vulnerability in the *gosu* executable, present in the database image. *Gosu* is a tool developed for docker and used to de-escalate privileges from root to a non-privileged user (Gravi, 2023). This tool is included in the default mariadb image as it's used to execute mysql using the mysql user during container startup. Unfortunately, the executable itself uses a vulnerable version of a golang package (Snyk, 2023). To mitigate this vulnerability, a newer version of gosu was downloaded and included in the stripped image. However, the newest version still uses the vulnerable image. This was confirmed by rescanning the image and reviewing the latest source files for gosu, which are available on Github.

2.3. Further Work - U2066664

The strip-image script which was used was important as it allowed to better understand and specifically select the needed files to be included in the stripped docker image, however was especially slow if it had a lot of supplied parameters. Alternatively, another tool called *Docker slim* would've been used as it is better maintained, provides more options for customisation and is significantly quicker at stripping docker images (Quest, 2023).

Furthermore, the current, stripped database image still includes several command line tools, such as chmod, chown, sed and others. Due to runtime security factors discussed later in the report, the functionality of a tool such as chmod is severely limited, however finding an alternative approach to remove these tools from the container after initialisation would further reduce the possible attack surface.

Finally, a docker registry for the company would be implemented, as it would provide strict management over the storage and distribution of in-house docker images. This would prevent the need to rely on images from a public registry for further development.

3. Runtime Hardening & Verification

3.1. Capabilities - U2136685

As containers don't contain, capabilities are needed to restrict the root user's ability when executing processes. When setting capabilities, the root user's ability can be limited to only the minimal privileges needed to run the processes required for a container to run successfully (Red Hat, 2022).

The capabilities for the web server and the database server were changed. Docker allows, by default, 14 capabilities enabled for all containers. The containers ran fine with the default capabilities, therefore we only had to test and choose out of these 14 capabilities (Docker, 2023b; Walsh, 2014).

The approach to setting capabilities was to use the "*--cap-drop=ALL*" flag when running a Docker container, then reading the logs to conclude which capabilities were required for the processes to run. For example, the database servers would fail when setting the "*--cap-drop=ALL*", then the log file would say that chown was not able to change permissions, therefore the flag "*--cap-add=CHOWN*" was added. Afterwards, the container would be run with the flags "*--cap-drop=ALL --cap-add=CHOWN*", then tested again and new capabilities were added until the container operated successfully and no unusual behaviour was found. The final capabilities for the database server were "*--cap-drop=ALL --cap-add=SETGID --cap-add=SETUID --cap-add=CHOWN*", whereas the capabilities for the web server were "*--cap-drop=ALL --cap-add=CHOWN --cap-add=NET_BIND_SERVICE --cap-add=SETGID --cap-add=SETUID*".

3.2. Volume Mounts - U2136685

Removing the database server's container removes the "iss2023db" database and its data along with it, therefore losing all the data accumulated from the website's users. This creates a problem, as if a container needs to be removed to, for example, change security options, all the data disappears. To fix this, there needs to be a persistent volume created that all the database containers can use.

Creating the volume can be done with the command "*docker volume create bravo-4-db-volume*". This volume is created in the Docker configuration files of the host system behind discretionary access control and is not accessible by a non-root user. This volume will need to be read from and written to, as database data is read and displayed to the web server and inserted from the web server, therefore when running the "docker run" command, the additional flag "*-v bravo-4-db-volume:/var/lib:rw,z*" is required to mount the volume to the "/var/lib/" folder, where the "mysql/" folder and database files will be created when importing the "iss2023db" SQL dump. Additionally, the "z" volume flag tells Docker that the volume is shared between containers, therefore keeping the SELinux labels the same between them so files are not blocked by the SELinux policies (Docker, 2023a).

The volume needs to be initialised with the database import as the "root" user, then run once as the "mysql" user so it can adjust the permissions to the "mysql/" folder in the volume.

3.2.1. Future Work - U2136685

In the future, adding read-only volumes for the web server and database servers would allow us to decrease the attack surface even further. Adding extra functionality to the web server in the future, such as file uploads, read-only filesystems would deter threat actors from, for example conducting remote code execution attacks, as we can control where files are uploaded and we can ensure that the location is read-only.

3.3. SELinux - U2136685

Security Enhanced Linux is a type of mandatory access control that implements labels that control the access of files, instead of the default discretionary access control that dictates who accesses the files by users and groups. SELinux labels everything on a machine, from processes to files and directories. SELinux uses policies implemented in the kernel that dictate what and who has access to these labelled objects (Walsh, 2014).

To enable Docker to run with SELinux, its systemd service needs to be modified. The service, found in "*/usr/lib/systemd/system/docker.service*", must be modified so that the flag "*--selinux-enabled*" is added to the "ExecStart" line. Then the system daemon needs to be reloaded, with "*sudo systemctl daemon-reload*" and the Docker service can be restarted, with "*sudo systemctl restart docker.service*". Docker will now obey the SELinux policies we will implement.

To find what needs to be allowed for the database server, creating a blank .te (Type Enforcement) file is needed. The policy can be set as permissive, to allow all access to go through and then the sealert process will show the SELinux alerts created by the database server container. The avcs denied can be seen in the audit.log file, analysed and selected to be allowed in the policy file. The "*sudo ausearch -m avc --start recent | audit2allow*" can be used to automate the process, though the policy should be reviewed as it can allow more things than necessary. To install the policy in the kernel, the .te file must be made into a policy file with the "*sudo make -f /usr/share/selinux/devel/Makefile <selinux-policy-textfile-name>.pp*", then installed with "*sudo semodule -i <selinux-policy-textfile-name>.pp"*. When no more errors are shown in the sealert browser, the policy can be enforced by commenting out the permissive line in the policy. If the enforcing status causes the database server to crash with no SELinux alerts, the "*sudo semanage dontaudit off*" command enables auditing for more general denials which can show additional avcs that need to be allowed in the policy. The same process is repeated for the web server as well.

Some non-critical access for the services in the Docker containers may be blocked. For example, in the database server container, mariadb tries to create a UDP socket but it is blocked by the policy. However, this does not impact the functionality of the database server, as seen in the figure below, therefore it does not need to be allowed in the policy.

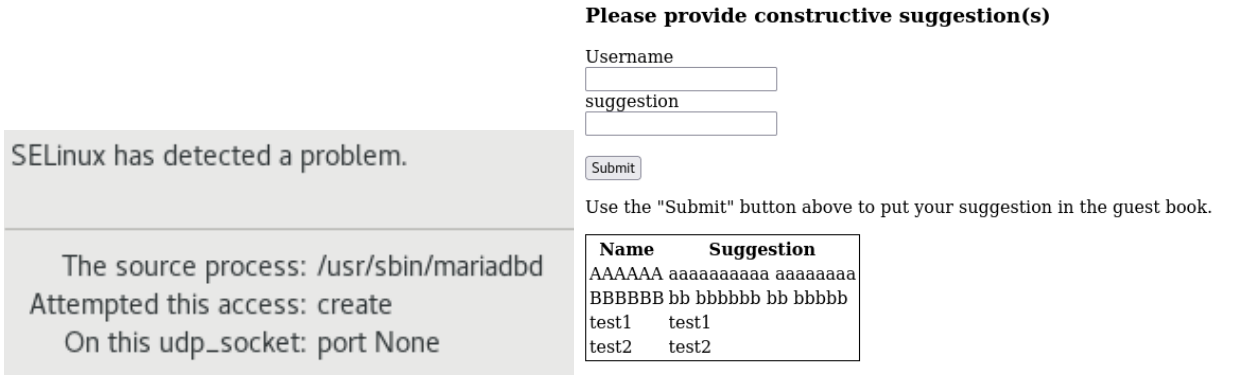

Figure 3.3.1.: Database working even when SELinux blocks some access.

3.4. Seccomp - U2156257

In addition to Selinux the runtime of the containers can be hardened by restricting the containers use of syscalls using a Seccomp profile. Docker has a default seccomp profile which allows the container to run, this profile is overly permissive and as such forms the basis of the refined profiles. The profile that has been created works by explicitly whitelisting certain syscalls whilst denying all others These syscalls form two categories; syscalls required by a docker container to start and syscalls required by the running process within the container. In order to find these syscalls a modified version of the supplied “build-minimal-syscalls.sh” script was created. The script individually removes one syscall from the default seccomp profile before creating a container with the modified profile, this allows a tester to find the syscalls that are required as without them the container would not start (Docker, 2020).

The script was modified by creating a test script in python that tests the functionality of the containers in three stages. First it verifies that the web server can be connected to and returns a 200 response code, second it verifies that the web server is connected to the database by checking that no connection errors were returned and thirdly it verifies that a post request can be successfully made. This test script can be incorporated into the syscall building script in order to create a container with one missing syscall, run the test script and if the script fails at any point the missing syscall is recorded as being necessary. These syscalls can then be used to create the seccomp profile for the web server and the database whilst also incorporating certain aspects of the default profile. There are two syscall generation scripts, one for the web server and one for the database but they only differ in which container is being run. Additionally the profiles can restrict the architectures that may be run on the containers, these were taken from the lists of supported architectures for the centos:7 images and the mariadb image which the container uses (CentOs Project, 2020; MariaDB developer community 2023).

In order to run the containers with the seccomp profiles active the following commands are added to the run command:

Web Server:
--security-opt seccomp:docker_webserver-service.json

Database:
--security-opt seccomp:docker_dbserver-service.json

3.4.1 Future Work - U2156257

In the future the seccomp profiles could be further restricted by restricting the arguments that are allowed to be invoked for each syscall or explicitly disallowing certain arguments from being used with certain syscall. This can be achieved using the args section for each individual syscall within the seccomp profile. This would further restrict the actions that can be taken on the container. However, the process to achieve such granularity is very long and not practical for inclusion as part of several web apps within a few months as each syscall must be investigated individually (Manning, 2017).

3.5. Non-Root Containers - U2136685

Docker containers usually run with their user as root. This allows them to run any commands they need and want to in order to launch the service they are required to do. A threat actor that compromises a container, that uses by default the root user, can gain control of all the other containers on the host computer. To make this more difficult for a threat actor, the container must be run as a non-root user. For example, the database server container can be run with the "*--user mysql*" flag. This ensures that all the initialisation is done as the mysql user, without root privileges. The database server can only run as the user "mysql" after it has been initialised and the /var/lib/mysql/ folder's permissions have been changed to mysql:mysql (after running the "*one-off-run-config-script.sh*" script).

For the web server container, this is more complicated, as to run nginx as a non-root user requires a lot of permission changing in the Dockerfile. A container can run as the user "nginx" when using the normal, full web image, however, due to the way the strip-image script works, the Dockerfile doesn't apply the crucial chown commands which change the ownership from the root user to the nginx user. The chown commands need to be done in the "./docker-entrypoint.sh" script. This script cannot be run as a non-root user as the user "nginx" does not have permissions on it. Moreover, the user "nginx" would not be able to change the ownership of files.

3.5.1 Future Work - U2136685

One may be able to run the nginx container as a non-user if a volume is first mounted and the ownership of all the files necessary are changed to nginx after first initialised, then the container can be run as nginx repeatedly. Moreover, the default port needs to be changed from 80 to a value above 1023, as ports 0 to 1023 are for privileged use only. However, due to time constraints this theory was not tested.

4. Additional Material

4.1. Read-only File System - U2136685

A read-only file system can be used for immutable containers, such as a web server but unlike a database server, which gets data written to it. In this case, the web server does not get anything written to its file system and it can simply be killed and removed if anything needs to be changed or if it is compromised. The container can be brought back up without a user noticing any changes. This was not implemented in the final build (Gills, 2018).

5. Evidence

App	*Interaction*	*Evidence*
Database	Connect with the web server.	Figure 5.1.1 shows that the two containers have been started with the run commands and the website can connect to the database
	Retrieve and store user input from the web server.	Figure 5.1.2 and 5.1.3 shows that the user's input is entered into the database and displayed on the website
	Run with the provided configuration files from the development team.	Figure 5.1.1 shows that the provided config files, when run, create a working container
	Run the container from a stripped image.	Figure 5.1.4 and 5.1.5 shows that the stripped image successfully runs a database container.
	Running with an enforcing SELinux policy while keeping full functionality.	Figure 5.1.1 shows that the SELinux policy is used and the database is functioning
	Running with the minimal amount of capabilities and syscalls while keeping full functionality.	Figure 5.1.1 shows that the seccomp policy and capability reductions have been applied as part of the run command and the containers still run
	Running with a read-only filesystem while keeping full functionality.	*This was not implemented in the solution.*
	Adding persistent volume so database data remains between containers.	Figure 5.1.6 shows that the database server data is persistent across different containers
Website	A user can access the website through localhost:8080.	Figure 5.1.1 shows that the two containers have been started with the run commands and the website can connect to the database
	Connect to the database to retrieve the stored data and display it to a connected user.	Figure 5.1.1 shows that the two containers have been started with the run commands and the website can connect to the database
	Take user input from the web interface and store it in the database.	Figure 5.1.2 and 5.1.3 shows that the user's input is entered into the database and displayed on the website
	Run the container from a stripped image.	Figure 5.1.4 and 5.1.5 shows that the stripped image successfully runs a website container.
	Running with an enforcing SELinux policy while keeping	Figure 5.1.1 shows that the SELinux policy is used and the web server is functioning

	full functionality.	
	Running with the minimal amount of capabilities and syscalls while keeping full functionality.	Figure 5.1.1 shows that the seccomp policy and capability reductions have been applied as part of the run command and the containers still run
	Running with a read-only filesystem while keeping full functionality.	*This was not implemented in the solution.*

Table 5.1.: Set of Adequate Interactions Evidence

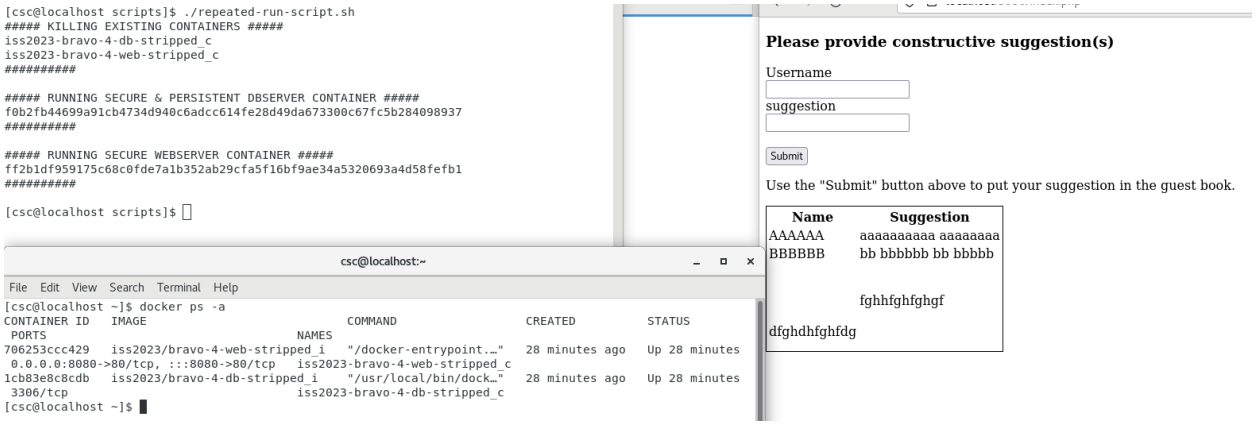

Figure 5.1.1.: Correct Operation of Containers

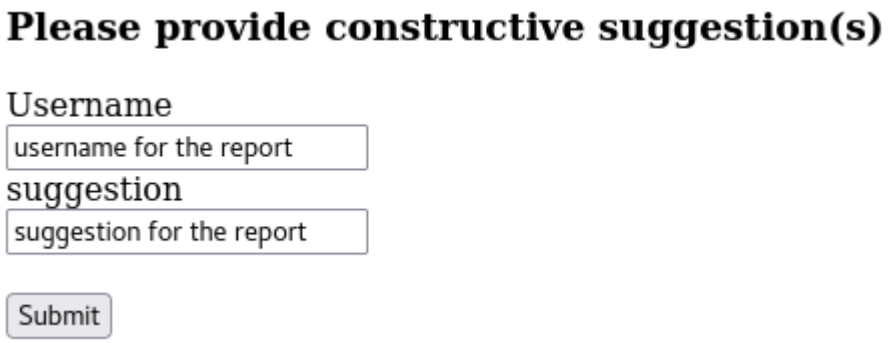

Figure 5.1.2.: Web Server input

Figure 5.1.3.: Database Communicated with the Web Server

```
CONTAINER ID   IMAGE                           COMMAND                  CREATED         STATUS         PORTS                                   NAMES
90497d85b5f3   iss2023/bravo-4-web-stripped_i   "/docker-entrypoint.…"   32 minutes ago   Up 32 minutes   0.0.0.0:8080->80/tcp, :::8080->80/tcp   iss2023-bravo-4-web-stripped_c
cad8fdf03211   iss2023/bravo-4-db-stripped_i    "/usr/local/bin/dock…"   32 minutes ago   Up 32 minutes   3306/tcp                                iss2023-bravo-4-db-stripped_c


REPOSITORY                      TAG       IMAGE ID       CREATED         SIZE
iss2023/bravo-4-db-stripped_i    latest    1b89d4df0b3d   5 hours ago     70.7MB
iss2023/bravo-4-web-stripped_i   latest    a0e94234672f   5 hours ago     24.7MB
```

Figure 5.1.4.: Stripped images and successfully running containers of the stripped images.

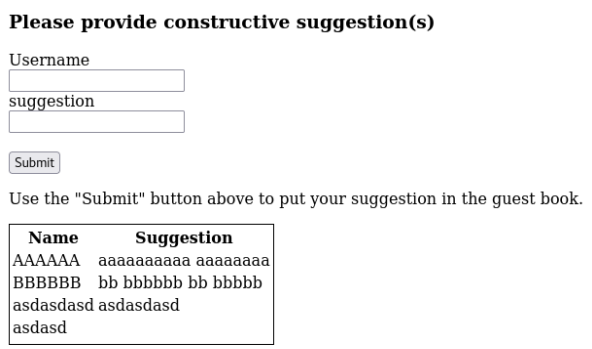

Figure 5.1.5.: Successfully run a website with stripped images.

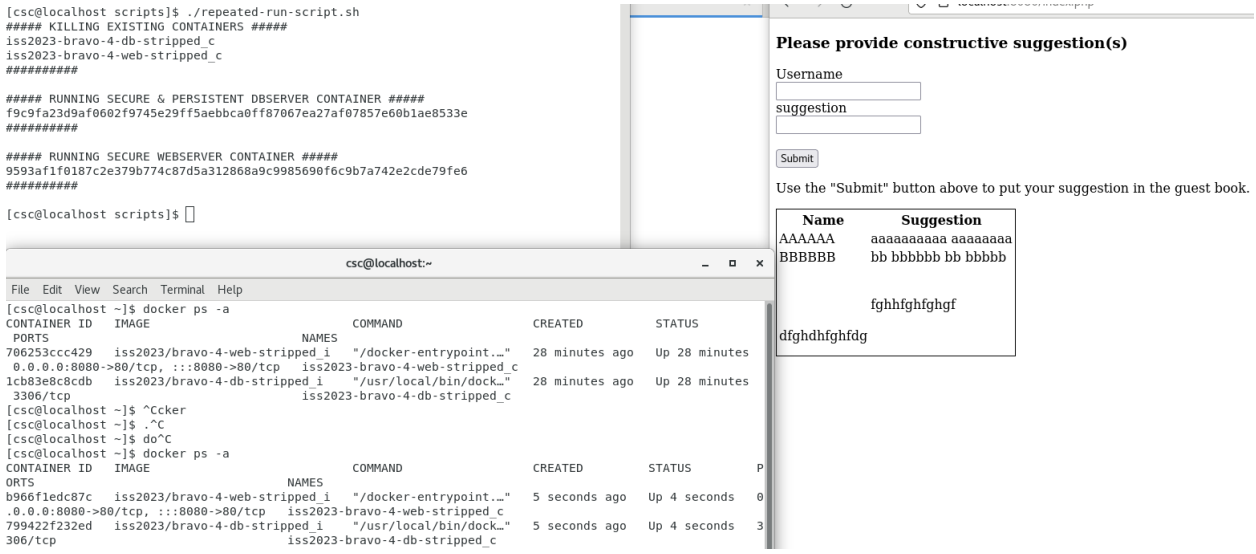

Figure 5.1.6.: Persistent database data after removing a container

6. References

CentOs Project (2020). *centos - Official Image*. [online] hub.docker.com. Available at: https://hub.docker.com/_/centos/ [Accessed 24 May 2023].

Docker (2020). *Seccomp security profiles for Docker*. [online] Docker Documentation. Available at: https://docs.docker.com/engine/security/seccomp/ [Accessed 24 May 2023].

Docker (2023a). *Bind mounts*. [online] Docker Documentation. Available at: https://docs.docker.com/storage/bind-mounts/#configure-the-selinux-label [Accessed 24 May 2023].

Docker (2023b). *Docker run reference*. [online] Docker Documentation. Available at: https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities [Accessed 24 May 2023].

Gills, A.S. (2018). *Is read-only mode a viable approach to container hardening? | TechTarget*. [online] IT Operations. Available at: https://www.techtarget.com/searchitoperations/answer/Is-read-only-mode-a-viable-approach-to-container-hardening [Accessed 24 May 2023].

Gravi, T. (2023). *gosu*. [online] GitHub. Available at: https://github.com/tianon/gosu [Accessed 24 May 2023].

Manning, M. (2017). *Docker Seccomp JSON Format*. [online] www.antitree.com. Available at: https://www.antitree.com/2017/09/docker-seccomp-json-format/ [Accessed 23 May 2023].

MariaDB developer community (2023). *Docker Hub*. [online] hub.docker.com. Available at: https://hub.docker.com/_/mariadb [Accessed 24 May 2023].

Moby Project (2022). *The Moby Project, default.json*. [online] GitHub. Available at: https://github.com/moby/moby/blob/master/profiles/seccomp/default.json [Accessed 20 May 2023].

Quest, K. (2023). *Optimize Your Experience with Containers. Make Your Containers Better, Smaller, More Secure and Do Less to Get There (free and open source!)*. [online] GitHub. Available at: https://github.com/slimtoolkit/slim [Accessed 24 May 2023].

Red Hat (2022). *Chapter 8. Linux Capabilities and Seccomp Red Hat Enterprise Linux Atomic Host 7 | Red Hat Customer Portal*. [online] Red Hat. Available at: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/linux_capabilities_and_seccomp [Accessed 24 May 2023].

Snyk (2023). *Snyk Vulnerability Database | Snyk*. [online] Learn more about Go with Snyk Open Source Vulnerability Database. Available at: https://security.snyk.io/vuln/SNYK-GOLANG-GOLANGORGXSYSUNIX-3310442 [Accessed 24 May 2023].

Walsh, D.J. (2014). *Bringing new security features to Docker*. [online] Opensource.com. Available at: https://opensource.com/business/14/9/security-for-docker [Accessed 24 May 2023].