Lab 4
Containerization, Orchestration and SELinux

Due: October 11, 2023

Please submit a separate report and submission zip/archive for the lab. Please make sure to include necessary screenshots, portions of code as necessary in the write-up to make the report a comprehensive document.

Part 0: Prerequisites

You'll be performing all steps of the lab on the Ubuntu VM image provided in the link. It is an extension on the previous Ubuntu image but has more compute, memory and storage which is needed for this part of the lab.

`https://drive.google.com/file/d/1739pCF5PzUsvsQmqPgsMeJj0KuWKQq-c/view?usp=sharing`

Part 1: Vulnerable Web-Apps

1a) Set up a web-service in a container

In this first section, we'll set up a web app with exploitable vulnerabilities in a container and use strace to identify the presence of those vulnerabilities.

1. Install Docker

   ```
   sudo apt update
   sudo apt install docker.io docker-compose
   ```

   You'll need to add your user to the "docker" group in order to run Docker without root privileges (i.e. without sudo). Execute the following command and restart your VM:

```
sudo usermod -a -G docker $USER
```

After rebooting, execute the command `groups` and you should see docker in the list.

2. Set up the Damn Vulnerable Web App container in Docker. You should follow the following guide from Docker on how to do this:

   `https://hub.docker.com/r/utspark/dvwa_class/`

   We recommend setting the difficulty to low.

3. PHP Injection

   Use the file upload portal to upload your own (malicious) PHP to the website. PHP is a server-side scripting language that is typically embedded in a static HTML page. The server will execute all PHP in the page before transmitting the final page to the client.

   Create a PHP file that does the following

   - prints out the path to the current directory,
   - prints the contents of the current directory,
   - prints the contents of the root of the filesystem at /,
   - and prints the number of processes running in the system.

   Please include the PHP file in your lab submission archive.

   The PHP function `shell_exec` should come in handy:

   - `https://www.php.net/manual/en/function.shell-exec.php`
   - The page above contains example code to get you started

   The following bash command will pring the number of processes running:

   - `ps aux --no-headers | wc -l`

   Once you upload the PHP file, DVWA will tell you the path where it was uploaded so that you can access it. When you navigate to that page, the server will execute your PHP. **Include the server's output in your report.**

- **Look at the contents of the root of your filesystem by running ls / in your VM. Does the server's view of the filesystem root differ in any way?**
- **What about the number of processes that the server thinks is** running?
- **Why might this be the case?**

4. Content Security Policy Bypass

   Navigate to the CSP Bypass tab. We leave it up to you to read the links under "More Information" and figure out how to exploit this vulnerability.

   You must leverage this vulnerability to execute Javascript that pops up an alert window with some text. **Include a screenshot of the popup window over DVWA in your report.**

   Describe how you exploited this vulnerability in your report. Include any source files in your lab submission archive.

5. SQL Injection

   Navigate to the SQL Injection tab. We leave it up to you to read the links under "More Information" and figure out how to exploit this vulnerability. You must leverage this vulnerability to extract the username and password of every user account in the database. Make sure to test your findings at the login prompt. **Include a table of the usernames and passwords in your report.**

   Describe how you exploited this vulnerability in your report. Include any source files in your lab submission archive.

6. Conclusion

   - **In what ways does containerizing the web app limit the attack surface? In what ways does it fall short?**

1b) Getting familiar with strace

`strace` is a tool used to profile the system calls being executed by a process.

- Read up on `strace`: `http://www.strace.io/`

`strace` attaches to processes and logs the system calls it makes, including arguments. You'll use `strace` to detect a DVWA exploit by looking for suspicious system calls. Before continuing, shut down the container you opened for the previous exercises. You can do this by simply entering Ctrl-C at the terminal running your container.
When you installed Docker, it ran a process on your system called `containerd`. You can read more about `containerd` here: `https://blog.docker.com/2017/08/what-is-containerd-runtime/` Essentially, `containerd` executes system calls on behalf of your container. Therefore, to monitor the system calls run by our web app, we will need to run `strace` on `containerd`. See if you can see `containerd` in your list of running processes:

- `ps -ef | grep containerd`

One line should look like the following:

```
root    651   1   0   09:17 ?      00:00:46 /usr/bin/containerd
```

`root` is the user that ran the process. `651` is the process ID (or PID) of `containerd`. `1` is the parent process. On your system, the PID will almost certainly be different. Make note of the PID of `containerd`, you'll need it next.
To monitor the system calls made by the Damn Vulnerable Web App, we will attach `strace` to the `containerd` process and log its system calls as well as the system calls of its child processes.
Read the help text of `strace` (`strace -h`) to understand its arguments. Then compose an `strace` command that will:

- Attach to `containerd` using its PID
- Log output to a file
- Trace all forked children of `containerd`

Once strace is attached, leave it running and start up your Damn Vulnerable Web App container:

```
docker run --rm -it -p 80:80 utspark/dvwa_class:2.0
```

Set up DVWA as before and navigate to the Command Injection tab. On this screen, you'll see a text box where you can enter an IP address to `ping`.

Behind the scenes, the server is quite literally executing:

```
ping <your input string here>
```

If we start our input string with a semicolon, we can close the ping command, and then the remainder of the string will be executed as a separate command. This means we can run arbitrary commands on the server! We will aim to identify this exploit using `strace`.

Inject the following bash command into the web app:

```
echo "malware" > /tmp/maliciousfile
```

Now open the log file produced by strace.

Identify the system calls in the log that was created and write to the file `/tmp/maliciousfile` and include those lines of the long in your report

You can use `grep` to search for the filename `/tmp/maliciousfile` in the log file.

1c) Limit who on the network can access the website using iptables

`iptables` is a command-line firewall utility that uses policy chains to allow or block traffic. When a connection tries to establish itself on your system, `iptables` looks for a rule in its list to match it to. If it doesn't find one, it resorts to the default action.
Reference: `https://www.howtogeek.com/177621/the-beginners-guide-to-iptables-the-linux-firewall/`

1. `iptables` should already be installed. If not, run `sudo apt install iptables`
2. Disable the existing firewall: `systemctl mask firewalld && systemctl stop firewalld`
3. Use the `iptables` command to allow only a single machine to access the web server on port 80. You should test it with your own ip address, but for submission you must only allow 10.157.90.8

Ubuntu iptables reference: `https://help.ubuntu.com/community/IptablesHowTo`

Include in your lab submission archive:

- A file containing your iptables rules. You can print out all rules with the command:

  `sudo iptables -S`

Part 2: SELinux

Please look through the below resources to get an introduction to SELinux before beginning the lab.

1. `https://wiki.gentoo.org/wiki/SELinux` Quick Introduction, Tutorials, etc.)
2. `https://docs-old.fedoraproject.org/en-US/Fedora/25/html/SELinux_Users_and_Administrators_Guide/index.html`
3. `https://wiki.gentoo.org/wiki/SELinux/Tutorials/Where_to_find_SELinux_permission_denial_details#SELinux_logging`

2a) Setup

1. Install SELinux and activate it:

   ```
   sudo apt install selinux-basics selinux-policy-default policycoreutils
   sudo selinux-activate
   ```

   Then reboot.

2. As root, run `sestatus -v` to verify that SELinux is enabled and currently in permissive mode.
3. SELinux will now audit file accesses on your system. You can view the audit log in /var/log/audit/audit.log and/or using journalctl.

2b) Create a simple policy module

In this problem you will create a simple policy module. You will write a simple C program that creates a file and writes to it. The policy will enforce that files created by that program have a specific label. Your program will also attempt to read from a secret file that it does not have access to.

1. Clone the `entnetsec` branch of the labs-sec repository from `https://bitbucket.org/utspark/labs-sec.git` into your home directory.

   ```
   git clone -b entnetsec https://bitbucket.org/utspark/labs-sec.git ~/labs-sec
   ```

2. Navigate to labs-sec/lab2/simple_example.

3. There are two files in the simple_module, please go through and understand them.

   (a) The file context simple.fc. In this file we associate the user, role, type, and level to each of the files the policy will manage, potentially using regular expressions.

   (b) The type enforcement simple.te. This file describes what is allowed by our policy rules.

4. Write a C program simple.c that does the following:

   (a) creates the file labs-sec/lab2/simple_example/data/simple.txt with the text "Hello World!" in it

   (b) reads and prints the contents of labs-sec/lab2/simple_example/data/secret.txt

5. Compile your program with `gcc -o simple simple.c`.

6. Now compile the simple policy module and load it by running `make && sudo semodule -i simple.pp`.

7. Run `sudo semodule -l | grep simple` and verify that the simple module is one of the policy modules in the system.

8. Now we need to apply the file context of our policy to label the files and directories. Run `restorecon -Rv labs-sec/lab2/simple_example` to recursively apply the labels.

9. Run `ls -Z` to observe that your simple executable has the `simple_exec_t` label.

10. Next we will add a systemd service to launch our program, otherwise it will inherit the context from our current unconfined process. Note you can see your current context with the command `id -Z`.

11. Copy the `simple.service` file to `/etc/systemd/system`.

12. You can now start the service with `systemctl start simple`.

13. Run `journalctl -xe` to see the logging for the simple service.

14. You should observe an audit error similar to this when your program attempts to read secret.txt:

    ```
    AVC avc: denied { read } for
    pid=3126 comm="simple" name="secret.txt" dev="sda1" ino=3277651
    scontext=system\_u:system\_r:simple\_t:s0
    tcontext=unconfined\_u:object\_r:user\_home\_t:s0
    tclass=file
    ```

15. Run `ls -Z` data and observe that simple.txt has been created with the `simple_var_t` type.

Include the following in your lab submission archive:

- simple.c source file
- simple.pp policy module
- Screenshot of `ls -Z` output from step 15
- Screenshot of your log denying access to secret.txt

Questions to answer in your report:

1. Explain the contents of simple.fc. What role does this file play in defining our SELinux Mandatory Access Control policy?
2. Do the same for simple.te.

Part 3:

3a) Orchestration with Kubernetes

We will learn about container orchestration and role based access control of services using Kubernetes. Kubernetes is used in industry extensively to develop and deploy production grade services and applications. In the previous section we had a docker container which ran the DVWA. The application has a web-server and a SQL Database both of which are running on the same container. In this section we will learn how to build those container images and run the server and SQL on different containers. We will see how to forward network ports from container to the host system so that these services can talk to each other.

Docker applications

Docker configurations are usually written in a Dockerfile. This docker file is then read and built into an image (`docker build -t <image>:<tag> .`). The image is then used to run an application (`docker run <image>:<tag>`). To simplify this process, we use `docker-compose` which reads docker files, builds the image and then runs it.

1. Clone the git repo to get all the necessary files for the lab:

   `git clone https://github.com/prateeksahu/simplePhpSQL_k8s.git`

2. Run the application using `docker-compose up`.

3. Open `http://localhost:8000` on the browser. Read the file under `src/index.php` and understand how the connection to the MySQL database is opened.

4. Understand the Dockerfile and tdovker-compose.yml files.

Some useful references:

- `https://docs.docker.com/compose/compose-file/`
- `https://docs.docker.com/engine/reference/builder/`

Answer the following questions:

1. What IP address and port does the web-service use to connect to the SQL DB? Refer to the source file `src/index.php` to find the answer. Explain what you see on the homepage `http://localhost:8000`.

2. Do necessary changes so that the web-server now serves at `localhost:9000`. Explain the change and give screenshots

So now when a request comes to the web-server running in a particular container, the server will connect to the DB to retrieve table entries. When in a production grade system, there can be thousands of requests coming in every second, often too many for a single web server to handle with reasonable latency. Consequently, developers want to be able to dynamically start multiple instances of a given web application to handle the quantity of traffic coming in at any given time. Scaling of applications dynamically is difficult when we use native docker. Kubernetes fills in this gap. For our labs we will use microk8s, which is a flavor of kubernetes which is fully supported

on Ubuntu. I urge everyone to use the given Ubuntu VM, since the rest of the lab is on microk8s and I cannot guarantee behaviour of it on Mac and Windows systems.

Install Kubernetes

1. `snap install microk8s --classic`
2. Similar to docker, we want to add the user to microk8s group to avoid repeated "sudo"s. You will need to reboot the VM afterwards

   `sudo usermod -a -G microk8s $USER`
3. `microk8s.enable registry dns dashboard`.

   Make sure all the pods are healthy and running before proceeding with the lab. Check status with `microk8s.kubectl get pods --all-namespaces`. In case you see containers in `ContainerCreating` or `Unknown` state, disable all (`microk8s.disable <service>`) and try enabling each of the services one by one instead of all together.
4. `sudo microk8s.enable rbac`

   This enables various features of kubernetes that we are going to use as a part of this lab.

   - **Registry** is the local repository where microk8s, builds/pulls images to.
   - **DNS**is the service which takes care of automatic hostname to container IP mapping for ease of DNS lookup. For kubernetes hostname is same as the service name.
   - **Dashboard** is a web-ui to see the health and deployment status of all the pods in a k8s cluster.
   - **RBAC** is a feature in k8s to have Role Based Access Control over the various Pods and/or the Cluster. It is used to provide access control security in production level application deployment.
5. We need to get the image that we built eariler to be available to microk8s, since microk8s cannot see images that we built using docker commands. Check your built images by `docker images`.
6. Push the web-service image to microk8s registry using the following commands:

```
docker tag <image-id> localhost:32000/<image-name>:k8s
docker push localhost:32000/<image-name>
```

7. Next we run the web-application in kubernetes:

   ```
   microk8s.kubectl apply -f webserver.yaml
   microk8s.kubectl apply -f webserver-svc.yaml
   microk8s.kubectl apply -f mysql.yaml
   microk8s.kubectl apply -f mysql-svc.yaml
   ```

   To delete or remove a service or deployment you can make use of the command `microk8s.kubectl delete -f <filename.yaml>`.

Answer the following questions:

1. Check the deployment of pods (containers) by `microk8s.kubectl get pods`. Check the service by `microk8s.kubectl get services`. You can get all pods and services by adding the keyword `--all-namespaces` to each of the above commands. Provide screenshots for both. What are the different namespaces you observe?
2. Explain the output of deployments and services. Where do we specify how many instances of each application is to be deployed?
3. Change the deployment to have 2 instances of web-servers and submit the screenshots.

Helpful Tips

You can ssh into a container by:

```
docker exec -it <conainer-name> /bin/bash
```

You can check all running docker containers by:

```
docker ps
```

You can connect to the MySQL container and execute queries using the MySQL command line by:

```
mysql --user=<username> --password
```

You can also ssh into kubernetes pod using the command:

```
microk8s.kubectl exec -n <namespace> pod/<pod-name> -i -t -- bash -il
```

You can print logs of any pod using the command:

```
microk8s.kubectl logs -n <namespace> pod/<pod-name> -v=9 --tail=20
```

RBAC

RBAC uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing admins to dynamically configure policies through the Kubernetes API.

Details of RBAC, Users, ServiceAccounts can be found on:

- `https://kubernetes.io/docs/reference/access-authn-authz/rbac/`

For the lab we will create a ServiceAccount and give it some permissions to see how that works.

1. Create a Service Account:

   ```
   microk8s.kubectl create serviceaccount user-sa --namespace default
   ```

2. Create a Role for this Service Account:

   ```
   microk8s.kubectl apply -f user-role.yaml
   ```

3. Bind the Service Account and Role:

   ```
   microk8s.kubectl apply -f sa-role-bind.yaml
   ```

4. Run Kubernetes Dashboard by exposing the dashboard service to an external port. You can edit the service by:

   ```
   microk8s.kubectl -n <namespace> edit service <service-name>
   ```

   Changing port type to NodePort or LoadBalancer instead of ClusterIP. Find the exposed port for the Dashboard service and access the Dashboard at `https://localhost:<port-num>` on a browser.

5. To login to the Dashboard we need Authorization tokens. Here we will use the service account we had created to access the service. We need to find the account token by running:

`microk8s.kubectl get secret`

and appending `--all-namespaces` or `-n <namespace>` for appropriate namespace account token names.

6. We get the token by trying to describe the secret associated with the appropriate token:

   `microk8s.kubectl describe secret <token-name>`

   We need to specify the namespace if the token name is associated with multiple namespaces.

7. We can login to the Kubernetes Dashboard using the token we find when we describe the token.

Answer the following questions:

1. On what port did you expose the dashboard service and how did you find it?

2. Explain the Dashboard when you login using the `user-sa` service account. Do you see all the pods that you see when you run `microk8s.kubectl get pods --all-namespaces`?
   Why or why not?

3. Create another service account which can access just the kube-system namespace. This service should have properties get, list, create, update & delete. Provide code and steps how you achieved this. Provide screenshots of the Dashboard.

3b) Creating a kubernetes cluster for DVWA

In the above section we saw how to build a simple web-app backed by a MySQL server into a kubernetes application. The next task is to build the DVWA we saw earlier into such a kubernetes application. When we have a single container containing all the components of the application, any vulnerability in one service exposes all other running services too. Using a kubernetes cluster, we cleanly separate this. We will learn how a fault-tolerant system can also be deployed using kubernetes. We need to clone the DVMA container github repo using `git clone https://github.com/opsxcq/docker-vulnerable-dvwa.git` and start by building the images. Following are top level guidelines which should help you in achieving your target.

1. Understand the Dockerfile and split it into separate Dockerfiles, one for the web-app and one for the database. You can get help from the Dockerfile and docker-compose files we had already created to achieve this step.

2. You might need to fiddle with the source code in the repository so that the web-app container now talks to the new container which runs the database.

3. Once you have built the appropriate images, edit the kubernetes deployment and service pods so that they deploy the new images. Access the DVWA application.

4. Start with just one instance of each of the services. Limit CPU and Memory for your web-app pods by using resource limiting features by kubernetes as described in `https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#meaning-of-memory`. We suggest a memory limit of at least 64Mi and compute limit of at least 100m. You can check the deployed pods' resource consumptions using `microk8s.kubectl top pods -n <namespace>`.

Answer the following questions:

1. All files used for the lab should be in the github repository. Mention the commands used to transfer the images into kubernetes registry in the README file. Create a commit of all the changes and submit the patch file of the change. You can create a patch by running the following command. This creates patch file between last two commits. If you have multiple commits, edit the git diff appropriately to encorporate all your changes.

   `git diff HEAD~1..HEAD > patch.diff`

2. Login to DVWA and try to crash the machine using a forkbomb attack. Try to access the webpage again. Does it work? Explain what happened. Show appropriate screenshots to backup your explanation.

3. Edit the web-app deployment to launch multiple instances of the service instead of just one. You might want to delete the existing web-app pod by either deleting using the deployment yaml or force delete of pod using `microk8s.kubectl delete pods -n <namespace <pod-name> --grace-period=0 --force`. Reapply the deployment yaml for the effects to take place.

4. Repeat the forkbomb and try to re-connect to the application. Does it work? Explain and provide appropritate screenshots. What could be the various DevOps use-cases of using kubernetes that you learnt from this experiment?

Feedback

We would like to get your feedback so that we can improve these labs in the future.

What did you like/dislike about this lab? Was it helpful in learning the material? Which sections were most/least helpful?