EE379K Enterprise Network Security Lab 2 Report

Student: Brian Cheung bc32427
Professor: Mohit Tiwari
TA: Antonio Espinoza
Department of Electrical & Computer Engineering
The University of Texas at Austin

September 25, 2019

Part 1 - Vulnerable Web Apps

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

For this lab, the Damn Vulnerable Web App (DVWA) was set up as a web-service in a Docker container using the following guide [1] and set to low difficulty.

PHP Injection

The objective of this section was to implement a PHP injection that printed out the path to the current directory, the contents of the current directory, the contents of the root of the file system, and the number of processes running in the system. This was done by uploading a PHP script (part-1/php_injection.php) to the DVWA and navigating to the file location. The following image1 is the output of the script:

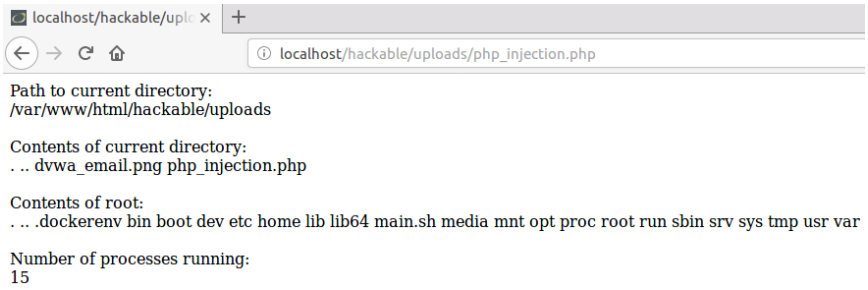

Figure 1: Screenshot of the server's output after executing the PHP script

Observations:

- When running `ls /` on the server and the VM, there are a few notable differences. The server shows some hidden files such as `.`, `..`, and `.dockerenv`, while the VM does not. However, the VM displays some files and directories that the server doesn't such as `cdrom/`, `initrd.img`, `initrd.img.old`, `lost+found/`, `core/`, `snap/`, `vmlinuz`, `vmlinuz.old`.
- Running `ps aux --no-headers | wc -l` from the VM's terminal resulted in 226 processes instead of the 15 processes when running the command on the server.
- These differences stem from the way Docker creates and manages containers. Docker uses cgroups to manage resource allocation and iso-

lates the container's processes from the host machine through the use of namespaces. Each container also has its own root file system, which explains the different views of the filesystem when running `ls /`. [2, 3]

1b - Getting familiar with strace

In order to detect a DVWA exploit, a tool called `strace` was used to log all the system calls made by the container. This can be done by attaching `strace` to the process called `containerd`, which executes the system calls on behalf of the Docker container. To attach `strace` to `containerd`, the PID of `containerd` must be passed as an argument to `strace`. To get the PID of `containerd`, the following command was run in the terminal:

```
$ ps -ef | grep containerd
root      1155     1  0 09:36 ?        00:00:08 /usr/bin/containerd
```

Then to monitor the system calls made by DVWA run the following command:

```
$ sudo strace -p 1155 -o strace.txt -f
```

Then run DVWA in a different terminal window:

```
$ docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

For demonstration purposes, a bash command was injected into the input in the 'Command Injection' tab:

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

The semi colon closes off the `ping` command making the server run the bash command that follows. The following lines identify the malicious sytem calls made during the exploit:

```
28552 execve("/bin/sh", ["sh", "-c", "ping  -c 4 ; echo \"malware\" > /t"...], [/* 9
...
28552 open("/tmp/maliciousfile", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
28552 fcntl(1, F_DUPFD, 10)             = 10
28552 close(1)                          = 0
28552 fcntl(10, F_SETFD, FD_CLOEXEC)    = 0
28552 dup2(3, 1)                        = 1
28552 close(3)                          = 0
28552 write(1, "malware\n", 8)          = 8
```

```
28552 dup2(10, 1)                       = 1
28552 close(10)                         = 0
28552 exit_group(0)                     = ?
28552 +++ exited with 0 +++
```

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

Part 2 - SELinux

Policy Modules

For this part of the lab, a simple policy module was created, and the file contexts were applied to the files and directories. Then running the `simple` executable, created a file, `simple.txt`, with the `simple_var_t` type. The following figure shows the output of running `ls -Z`:

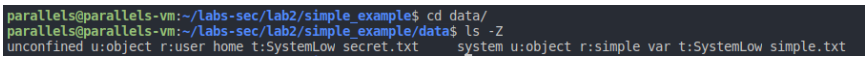

Figure 2: Screenshot of the context of simple.txt

The following figure shows the error received when the `simple` executable tries to read `secret.txt` without the correct required permissions:

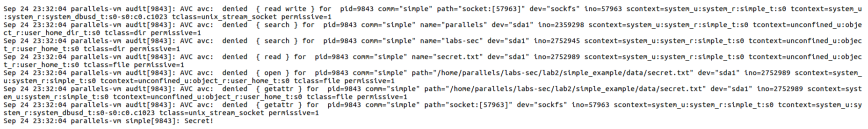

Figure 3: Screenshot of log denying access to secret.txt

Context files (simple.fc) "declare the security contexts that are applied to files when the policy is installed" [4]. Such security contexts include user, role, type, and level. Type enforcement files (simple.te) define types and assign rules to each of the types. Files associated with a specific type will have the permissions defined for that type.

Conclusion

The lab took about 40 hours which was a bit longer than expected. It was pretty interesting learning about socket connections in different languages

and how GET requests are built using socket connections. I think some parts of the lab were a little unclear and needed further clarification. Overall, this lab served its purpose in providing a more hands-on experience that helped improve my understanding of networking.

References

[1]

[2] J. Skilbeck, "Docker: What's Under the Hood?," January 2019.

[3] S. Grunert, "Demystifying Containers - Part I: Kernel Space," March 2019.

[4] "Red hat enterprise linux 4: Red hat selinux guide."