EE379K Enterprise Network Security Lab 2 Report

Student: Sean Wang, szw87
Professor: Mohit Tiwari, Antonio Espinoza
Department of Electrical & Computer Engineering
The University of Texas at Austin

September 25, 2019

Part 1 - Vulnerable Web-Apps

1a - Containers

PHP Injection

The Damn Vulnerable Web App in Docker was setup on low difficulty by following the guide [1]. Then, a PHP file (`part1/injection.php`) was uploaded to the web app, and could then be loaded and executed when navigated to. This PHP script prints the path to the current directory, the contents of the current directory, the contents fo the root, and the number of processes running in the system. The output of the script onto the webpage is shown below:

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

Contents of current directory:
. .. dvwa_email.png 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:
17
```

The server's view of the filesystem has a few differences from running `ls /` from the VM's terminal. For example, it shows files that are normally hidden, such as `.`, `..`, and `.dockerenv`. The files and directories it doesn't show that `ls /` shows include `cdrom/`, `initrd.img`, `initrd.img.old`, `lib32/`, `libx32/`, `lost+found/`, `snap/`, `swapfile`, `vmlinuz`, `vmlinuz.old`. Additionally, running `ps aux --no-headers | wc -l` from the VM's terminal resulted in 223 instead of 17. This difference is due to how Docker creates and manages containers. Docker utilizes the Linux kernel's cgroups and namespaces in order to manage and monitor resource allocation, and utilizes the C library, `runC`, that gives each container its own root file system, similar to a `chroot jail` [2, 3]. As such, the root of the filesystem seen by the Docker container and the VM are not the same filesystem, explaining the difference in output of `ls /`. Additionally, through the use of namespaces, Docker has isolated the processes inside the container from the processes of the VM running Docker.

Content Security Policy Bypass

On the Content Security Policy Bypass (CSP Bypass) page, there is a textbox that allows external scripts from certain allowed sites to be run. In order to execute Javascript that creates a popup alert, a simple line of Javascript was inserted into a pastebin:

```
alert("this is a popup window");
```

Then, by inserting the pastebin link to the raw text, `https://pastebin.com/raw/wENwXfBR`, and pressing the include button, a popup was generated, like in Figure 1.

SQL Injection

On the SQL Injection tab, there were several steps needed to get the right information to access all the login credentials. First,

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

was entered to check if this would return all records that are false and all that are true, as well. The results are shown in Figure 2, confirming that we can use the the previous input as a prefix to other inputs that query the rest of the information. For example, the next input was

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

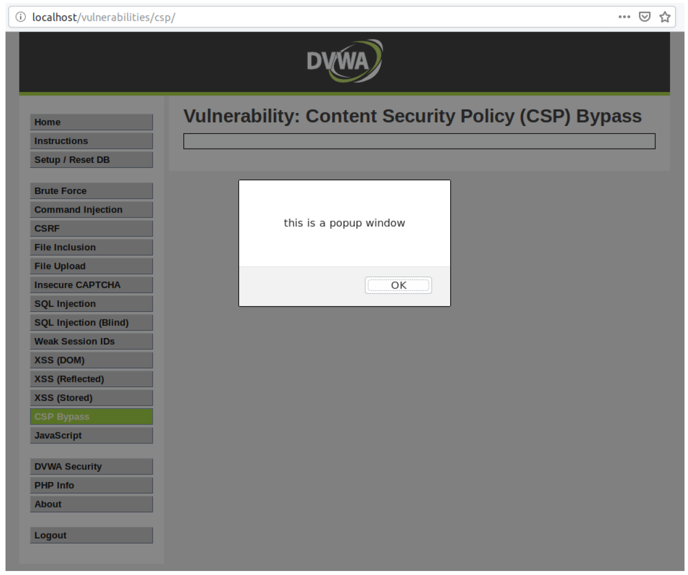

Figure 1: Javascript popup from CSP vulnerability

which returned information about all the other databases the server maintains, as shown in Figure 3. Next, changing the input to

```
%' and 1=1 UNION SELECT null, table_name FROM
  INFORMATION_SCHEMA.tables WHERE table_name LIKE 'user%' #
```

returned and confirmed the table name needed to find login credentials, `users`, shown in Figure 4. To get the credentials, the input was changed to

```
%' and 1=1 UNION SELECT null,
  concat(0x0a,user,0x0a,password) from users #
```

The results of this are shown in Figure 5 and include the username and hashed password at the bottom of each entry.
Using a website [4] to decrypt the hashes, the login credentials of every user can be determined, as shown in Table 1. Since the input wasn't being

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

Table 1: Table of credentials

sanitized, SQL queries could be submitted through the input box and then

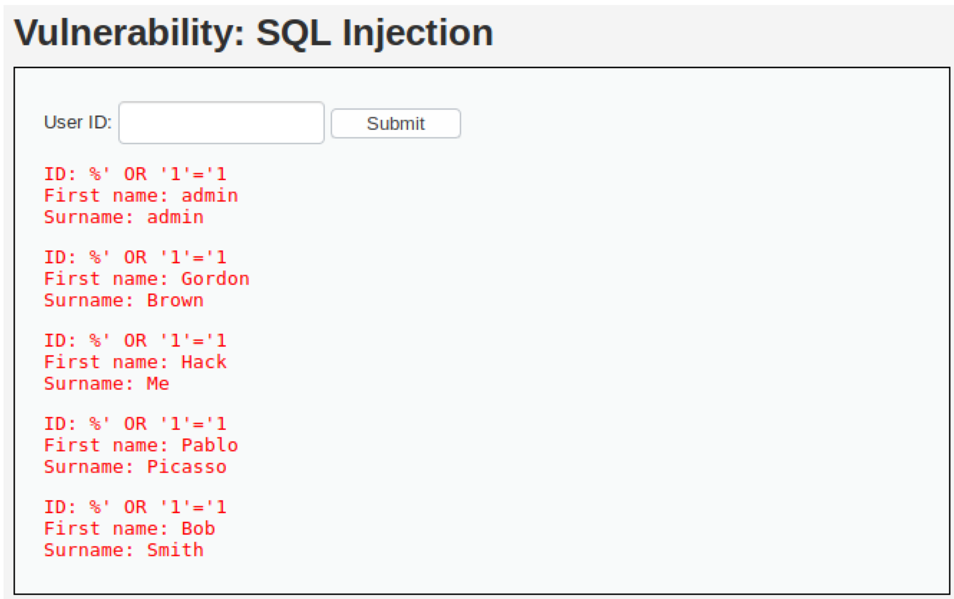

Figure 2: First SQL injection to check sanitization

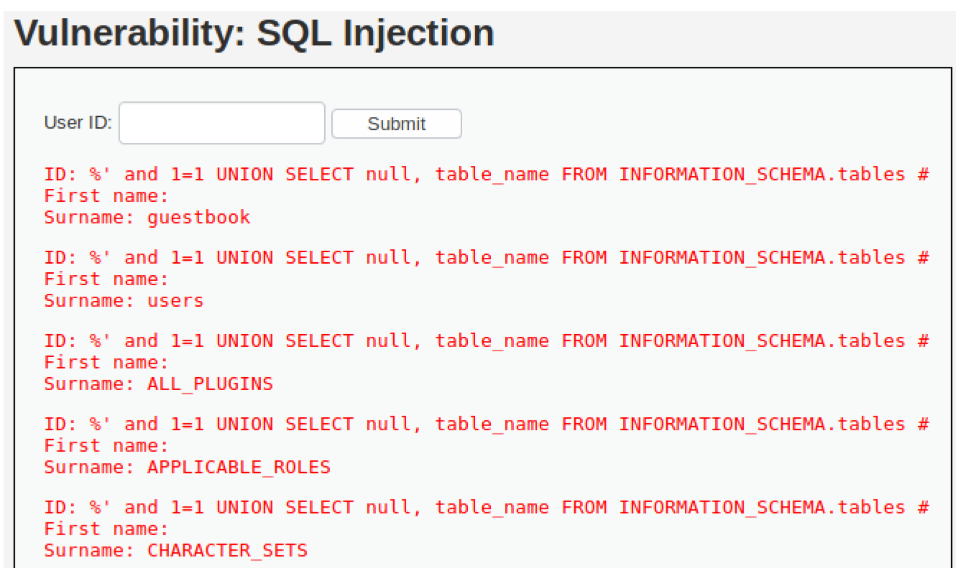

Figure 3: Result of query for table names (not all results shown)

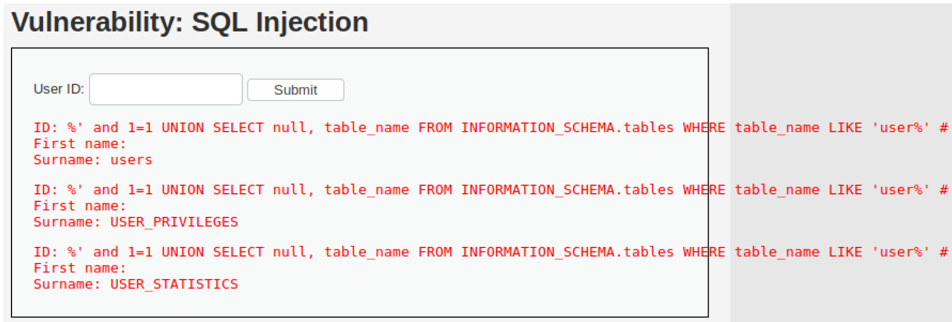

Figure 4: Result of query for table names similar to 'user'

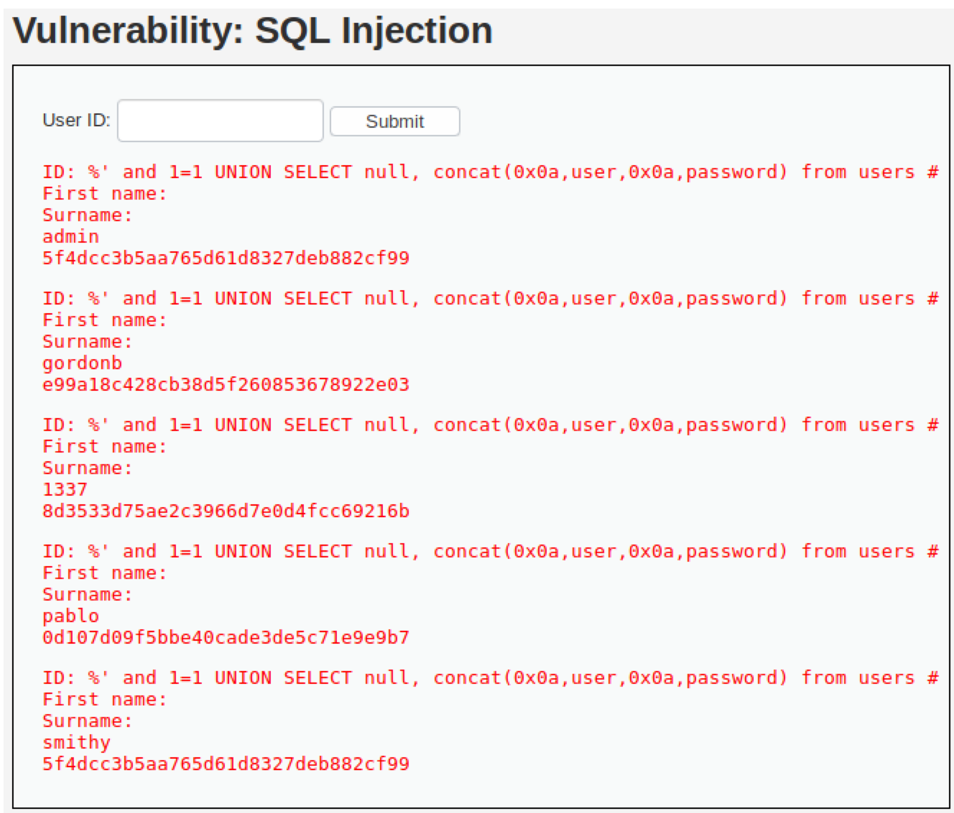

Figure 5: Result of query for user credentials

executed, returning all kinds of information from the database. The queries at the beginning were to test what was valid and figure out the internal SQL query that was being performed by the web app. Then, the table names could be queried using the `INFORMATION_SCHEMA` table to find the table name that was most likely to have user credentials. Once the right table is found, it becomes fairly simple to get the hashed passwords and then decrypt it using a tool. The end result is the exposure of all these users' credentials.

Containerizing the web app limits what the attack can see and modify since the server is isolated from the actual host machine. This is shown in the PHP injection where the results of `ls /` and `ps aux --no-headers | wc -l` are different when run from the PHP script on the server and from the terminal of the VM hosting the container. However, containerizing doesn't prevent the server itself from the kinds of attacks performed by bypassing the Content Security Policy and SQL injections. In other words, the machine hosting the container is protected from attacks coming from inside the container, but the inside of the container is still vulnerable.

1b - strace

A method to detect exploits performed on DVWA is to use the tool `strace`. It can attach to a process and log any system calls that process makes. To monitor any system calls made by DVWA, `strace` needs to be attached to a process called `containerd`. To get the PID of the process:

```
$ ps -ef | grep containerd
root      700     1  0 10:39 ?      00:00:00 /usr/bin/containerd
```

Then, to attach `strace` to the `containerd` process and all of its children and run DVWA:

```
$ sudo strace -p 700 -o strace.txt -f
$ docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

Then, for example, the following command can be executed into the 'Command Injection' tab by closing off the `ping` command with a semicolon:

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

The log produced from `strace` in `part1/strace.txt` can then be searched for system calls that create and write to that file:

```
8545  execve("/bin/sh", ["sh", "-c", "ping  -c 4 ; echo
      \"malware\" > /t"... ], 0x7ffd96384018 /* 9 vars */
```

```
       <unfinished ... >
...
8545   open("/tmp/maliciousfile",
       O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
8545   fcntl(1, F_DUPFD, 10)           = 10
8545   close(1)                        = 0
8545   fcntl(10, F_SETFD, FD_CLOEXEC)  = 0
8545   dup2(3, 1)                      = 1
8545   close(3)                        = 0
8545   write(1, "malware\n", 8)        = 8
8545   dup2(10, 1)                     = 1
8545   close(10)                       = 0
8545   exit_group(0)                   = ?
8545   +++ exited with 0 +++
```

1c - Limiting network access with iptables

First, `firewalld` is masked and stopped to prevent it from being started by other services and tells the system to use `iptables`' rules. This is done using the following command:

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

Now the `iptables` rules can be modified to only allow a single connection to the web server from `10.157.90.8`. This can be done with the following:

```
$ 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
```

Now, any connections not from the specified IP are dropped. The new `iptables` rules can be seen using:

```
$ sudo iptables -S
```

Alternatively, the output of `iptables-save` can be saved into a file, which contains the rules as well, like in `part1/iptables.txt`.

Part 2 - SELinux

For this part, a simple policy module was created and the file contexts were then applied. Running the `simple` executable will create a file, `simple.txt`, and attempt to read from a file, `secret.txt`. After compiling and loading the policy module and using `restorecon` to apply the labels, the `simple` executable now has the label `simple_exec_t`. Then, after starting the `simple.service` and looking at the logs, the following error can be seen when the program attempts to read `secret.txt` in Figure 6. Additionally, `ls -Z`

```
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { getopt } for  pid=5078 comm="(simple)" scontext=system_u:object_r:unlabeled
_t:s0 tcontext=system_u:system_r:init_t:s0 tclass=unix_stream_socket permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { setopt } for  pid=5078 comm="(simple)" scontext=system_u:object_r:unlabeled
_t:s0 tcontext=system_u:system_r:init_t:s0 tclass=unix_stream_socket permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { getattr } for  pid=5078 comm="(simple)" scontext=system_u:object_r:unlabele
d_t:s0 tcontext=system_u:system_r:init_t:s0 tclass=unix_stream_socket permissive=1
Sep 25 14:09:29 class-VirtualBox audit[581]: AVC avc:  denied  { use } for  pid=581 comm="gdbus" path="anon_inode:[eventfd]" dev="anon_inodef
s" ino=9854 scontext=system_u:system_r:policykit_t:s0 tcontext=system_u:object_r:unlabeled_t:s0 tclass=fd permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { shutdown } for  pid=5078 comm="(simple)" scontext=system_u:object_r:unlabel
ed_t:s0 tcontext=system_u:system_r:init_t:s0 tclass=unix_stream_socket permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { setrlimit } for  pid=5078 comm="(simple)" scontext=system_u:object_r:unlabe
led_t:s0 tcontext=system_u:system_r:init_t:s0 tclass=process permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { getcap } for  pid=5078 comm="(simple)" scontext=system_u:object_r:unlabeled
_t:s0 tcontext=system_u:system_r:init_t:s0 tclass=process permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { use } for  pid=5078 comm="simple" path="/dev/null" dev="devtmpfs" ino=6 sco
ntext=system_u:system_r:simple_t:s0 tcontext=system_u:object_r:unlabeled_t:s0 tclass=fd permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { search } for  pid=5078 comm="simple" name="class" dev="sda1" ino=557128 sco
ntext=system_u:object_r:unlabeled_t:s0 tcontext=system_u:object_r:user_home_dir_t:s0 tclass=dir permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { read } for  pid=5078 comm="simple" name="secret.txt" dev="sda1" ino=671633
scontext=system_u:object_r:unlabeled_t:s0 tcontext=system_u:object_r:user_home_t:s0 tclass=file permissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { open } for  pid=5078 comm="simple" path="/home/class/labs-sec/lab2/simple_e
xample/data/secret.txt" dev="sda1" ino=671633 scontext=system_u:system_r:simple_t:s0 tcontext=system_u:object_r:user_home_t:s0 tclass=file pe
rmissive=1
Sep 25 14:09:29 class-VirtualBox audit[5078]: AVC avc:  denied  { getattr } for  pid=5078 comm="simple" path="/home/class/labs-sec/lab2/simpl
e_example/data/secret.txt" dev="sda1" ino=671633 scontext=system_u:system_r:simple_t:s0 tcontext=system_u:object_r:user_home_t:s0 tclass=file
 permissive=1
```

Figure 6: Error logs from attempting to read from secret.txt

shows that the file `simple.txt` has been created with the `simple_var_t` type, shown in Figure . The context file, `simple.fc`, specifies what security

```
class@class-VirtualBox:~/labs-sec/lab2/simple_example$ ls -Z data/
 system_u:object_r:user_home_t:s0 secret.txt  system_u:object_r:simple_var_t:s0 simple.txt
```

Figure 7: Labels attached to simple.txt

contexts are applied when a policy is installed, including role, user, type, and level. This is how restrictions were applied to what the `simple` executable could modify. On the other hand, the `simple.te` file is a type enforcement file that defines certain types and what rules apply to those types. This then applies the rules and permissions associated with certain types to files of that type.

Conclusion

This lab did not take an unreasonably long time, rather it took a reasonable amount of time. This was also a very interesting lab, especially part 1, involving finding vulnerabilities in DVWA. It also brought up important topics to think about regarding how to prevent these kinds of vulnerabilities.

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."