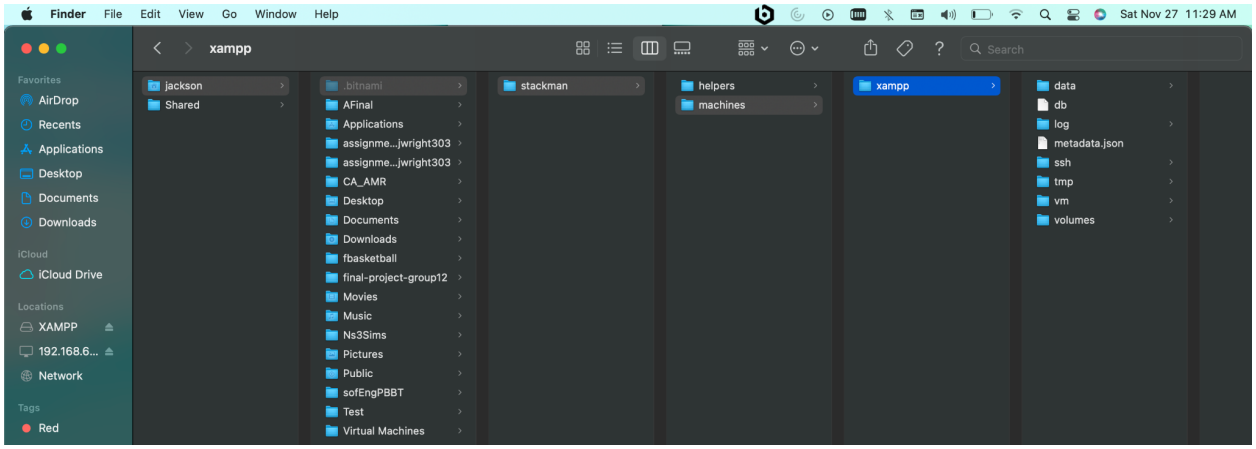
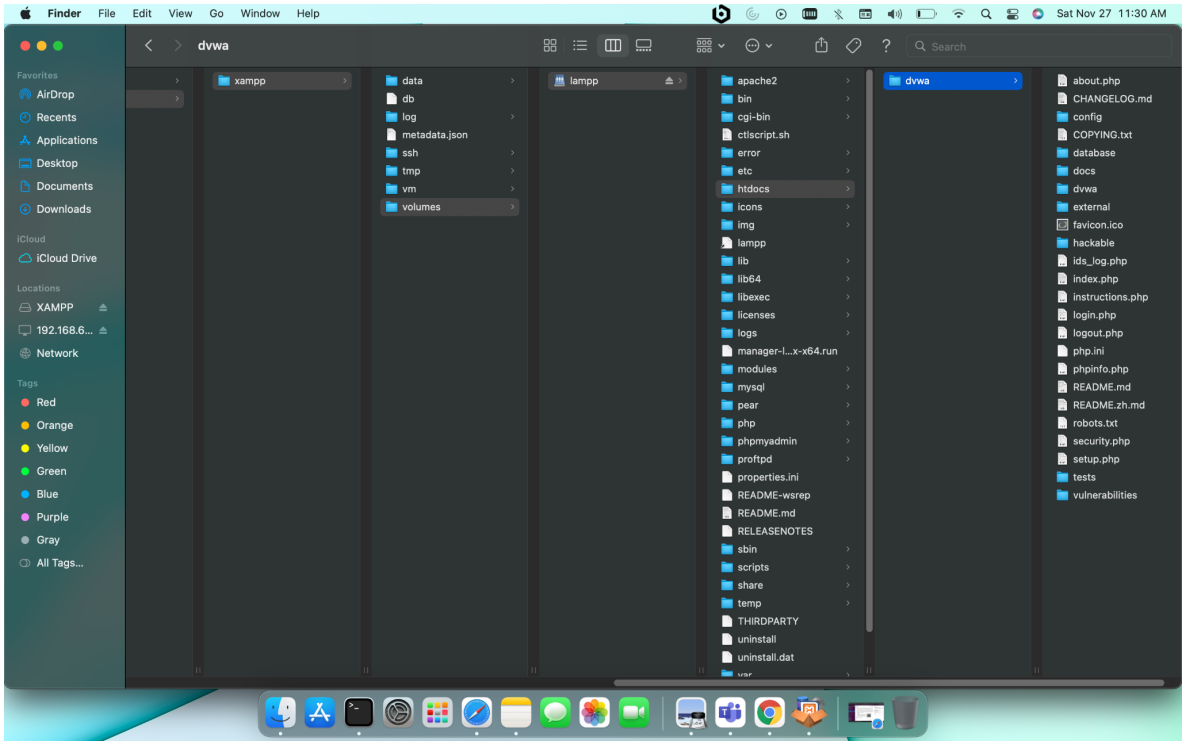
Setup

I used this guide for installation: https://www.youtube.com/watch?v=cak2lQvBRAo
The technology that I used for the setup was xampp which included the installation of an apache web server, and a MySQL database. The following are screenshots of my setup.

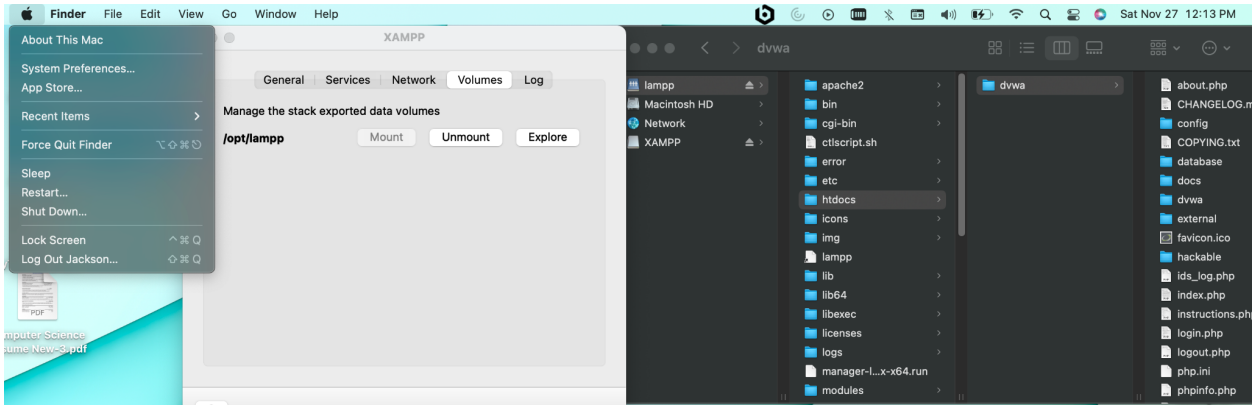





```
Jacksons-MBP:dvwa jackson$ pwd
/Users/jackson/.bitnami/stackman/machines/xampp/volumes/root/htdocs/dvwa
Jacksons-MBP:dvwa jackson$
```

Note: The installation of xampp on mac is extremely different from the installation on Windows. To start, when running the application for the first time there is no ability to choose where things are installed. For this reason, and to keep things simple, I decided to leave everything downloaded on my local machine. Lastly, to be able to even view these files I had to go to the

volumes section of xampp and select mount and explore. This created a lampp(?) directory which is where all the content of the webserver is stored. Below is a screenshot of how I was able to finally access the server contents and import the dvwa folder.



Vulnerability 1: Brute Force Login

The login functionality normally works by having users enter their username and password into the site. If the proper credentials are entered then the user is welcomed to the site, otherwise, an error message is given telling them the username and password are not recognized.

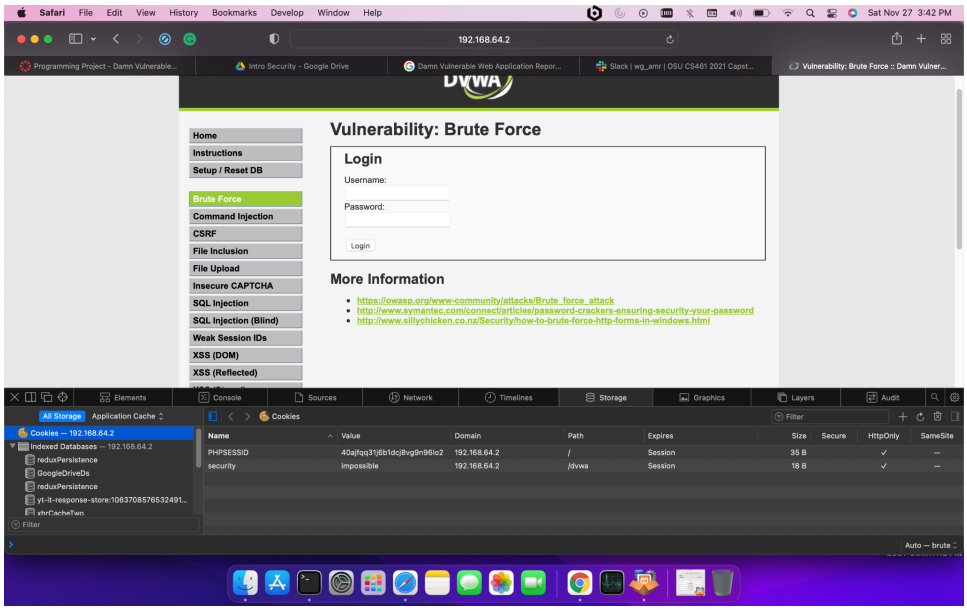
To exercise this vulnerability you need to have a username list and password list as well as a program to automate the login attempts. The program then iterates through the usernames and passwords and checks them against the site. All the user will have to do is look at the difference in the response message from the site (Incorrect user name versus a welcome message).

It's not necessarily that the feature worked differently than normal use, rather the feature didn't prevent malicious use. In fact, the problem is that the feature worked as normal for all the attempts which allow malicious users to test out a lot of different combinations to find a valid username and password combination

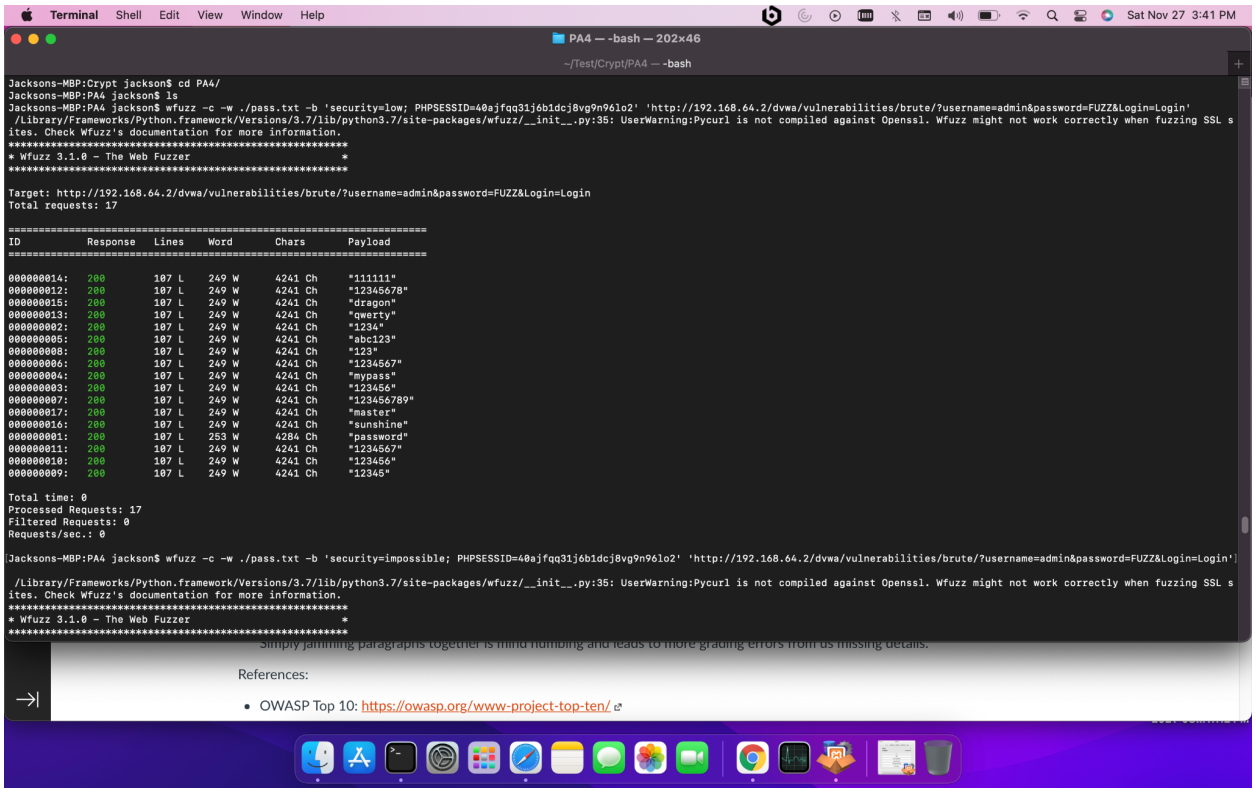
This vulnerability is really important because if malicious users can access other accounts then they can do a lot of damage. They could view the personal information of the account they logged into, or make changes to their account. Similarly, since they guessed the password they could change it so that the real user can't log in to their account. Furthermore, since we were able to get access to the admin account, we could potentially make significant changes to the site, including injecting malware, or a keylogger to steal other information.

To fix the vulnerability there could be a timeout function that locks an account for a fixed amount of time after too many login attempts. Similarly, for the case of the attack I used, special precautions could be made to give a similar message for successful and unsuccessful attempts (more on that later). I found some of these countermeasures by switching to the impossible security mode and using the same method.
Source:https://owasp.org/www-community/attacks/Brute_force_attack

Screenshots:



First I gathered the cookies information, mainly the PHP session ID



Notes: The process started by gathering the cookie information and URL from the session on the site. Then a password list was created (I made it myself using some random password and the one I knew was on there, but longer ones exist for this). Finally, I ran the wfuzz algorithm supplying the cookie information, password list, and URL. This printed out the response page for each session. I used the difference in word count to identify the correct password.

The OWASP vulnerability this relates to is number 7: Identification and authentication failure. This is because it failed to properly authenticate me since I was not the proper user logging in.

Source: https://www.youtube.com/watch?v=SWzxoK6DAE4

To automate these kinds of tests you first have to get password and username lists. Alternatively, you could do some probing until you find a valid username and just find a password list. For each username, you iterate through the password list, and for each, you attempt to access the site given the known URL (replace USER, PASS with username and password). Note you will have to pass the cookies to the site before attempting. (http://192.168.64.2/dvwa/vulnerabilities/brute/?username=USER&password=PASS&Login=Login). For each attempt, you record the response page, including the word count, lines, and characters. You will know there is a vulnerability when there are a difference in word count, lines, or characters. This is because there will be differences in the response message based on successful or unsuccessful access.

Vulnerability 2: Command Injection

This feature normally works by allowing users to ping devices given a specific IP address. This just sends a packet to the device, hopping from router to router, and a packet is sent back if it reaches the target notifying it has been received. This then prints out the time it took for the response of each packet.

To exercise this vulnerability all you need to do is add a semicolon after the IP address, as well as the other command you would like for it to run. An example of this would be “127.0.0.1; ls”. This makes it so that after it pings that IP address, it prints the names of files in the current directory.

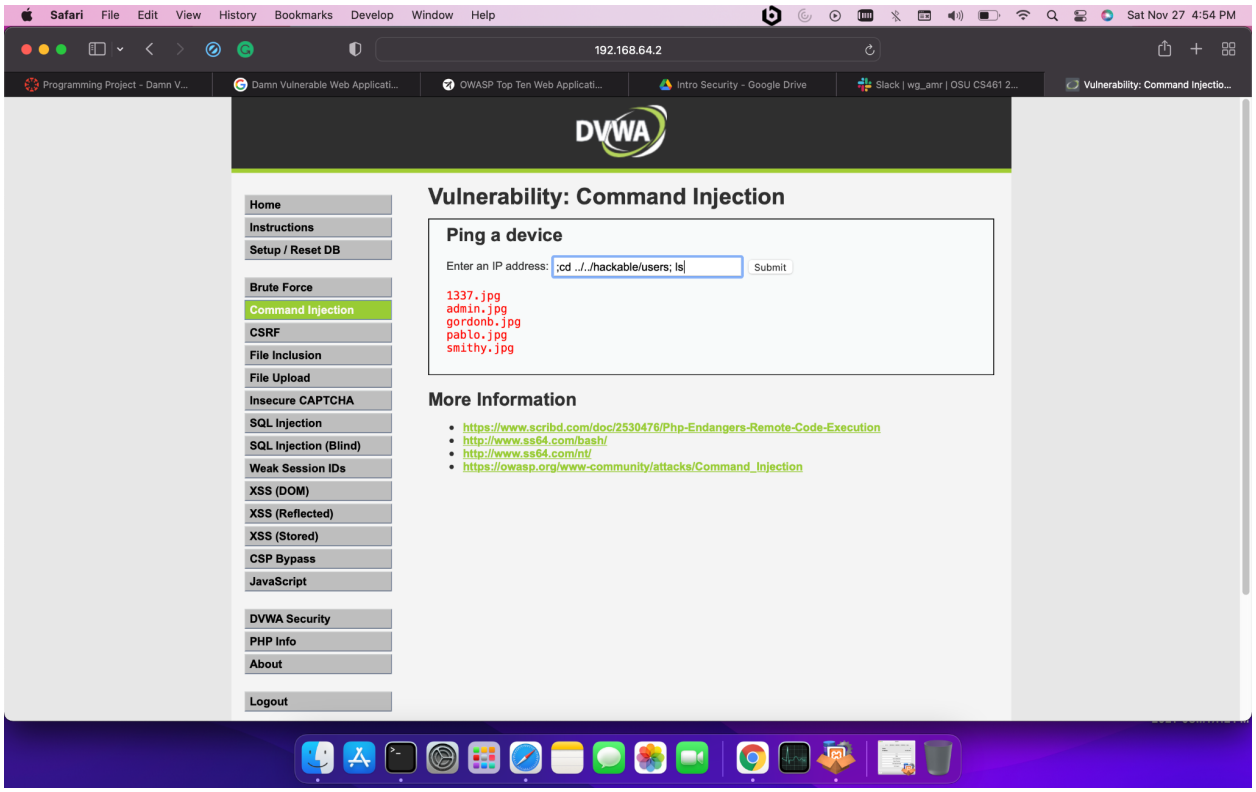
This behaves differently than normal in that it allows you to execute multiple different commands other than ping. In other words, it allows users to execute unauthorized commands such as ls.

The reason this works differently is because of the semicolon. Bash sees the semicolon as a separation of commands so after ping is done, it tries to execute the next command found after the semicolon. This also works differently because the input from the user is never checked.

This vulnerability is important because it allows users to gather a lot of information about the backend of the server. Similarly, since any commands could be run, an adversary can use this to create a reverse shell which could allow them to have significant access to the server. In my case, I was able to find the usernames of different users which I could use in combination with the first vulnerability to gain access to their accounts as well.

One way to prevent attacks like this is to not allow the OS to execute anything that is supplied by the user. This prevents commands that they give from being run. Similarly, these attacks can

be prevented by checking the input from the user. Making sure only numbers and periods are supplied in this case would prevent many commands from being executed by the system. https://portswigger.net/web-security/os-command-injection



This is an example of one of the commands I used which displayed different usernames of the site.

This corresponds to the third OWASP vulnerability: injection. This is because users are able to inject their commands and have them run by the site.

Source: https://www.youtube.com/watch?v=WiqRvlN_UIU

To automate these kinds of tests you could create a file with commands similar to what I used above. Similar to the brute force method, you could repeatedly enter the page, each time entering a new command. The word count of the response page could then be used to determine if behavior other than what is expected is produced. If you had access to the actual source code of the page you could do something similar using a file with a bunch of different commands and send each of them as input to the section. If any of them allow execution then you know you have found a vulnerability.

Vulnerability 3: Cross Site Request Forgery

This function normally works by allowing users to change their password to something new. Here you enter in the new password you twice to confirm the new password you would like to change it to.

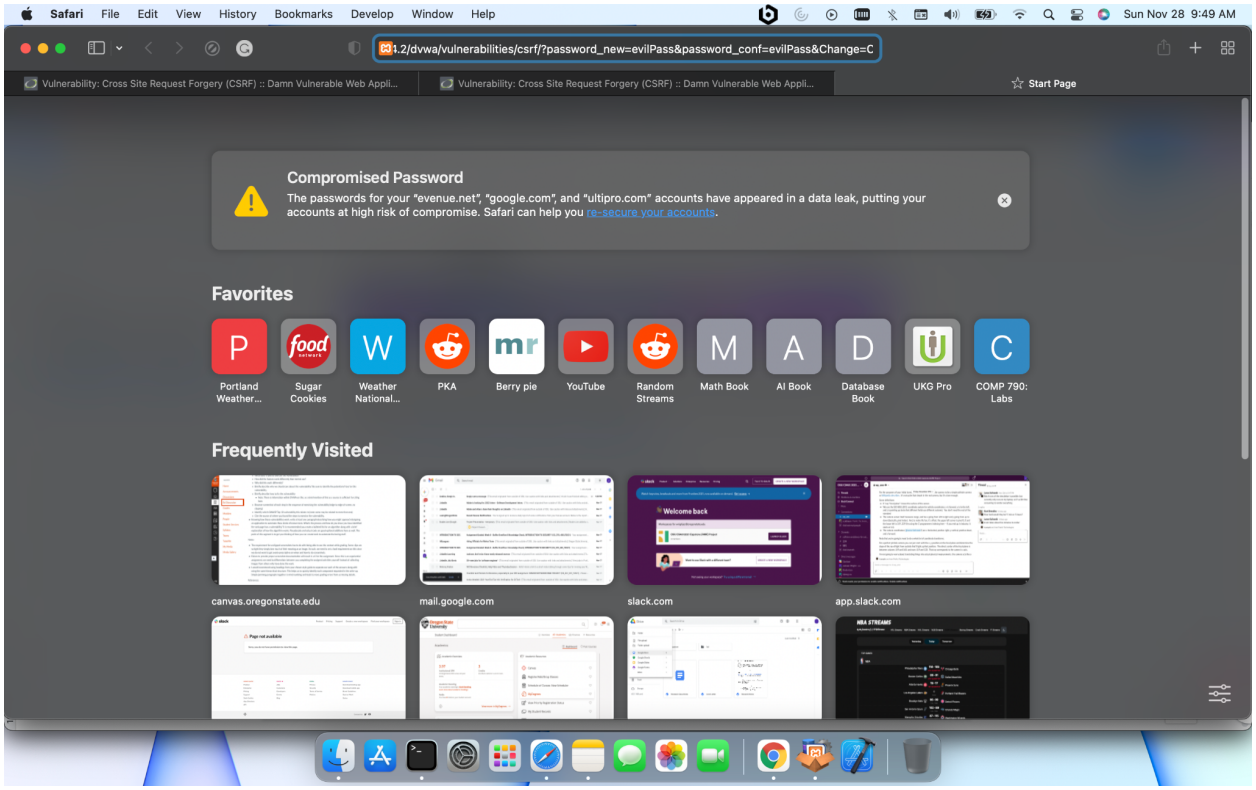
All it takes to exercise this vulnerability is the url of the website. If an adversary was able to get a link similar to what is shown below they could change the password_new and password_conf parameters to a password they want it to be. After this they would just need to send the link to the actual user of the site in a way that gets them to click on it. Once the user clicks the link, the information is automatically imputed into the form changing their password.

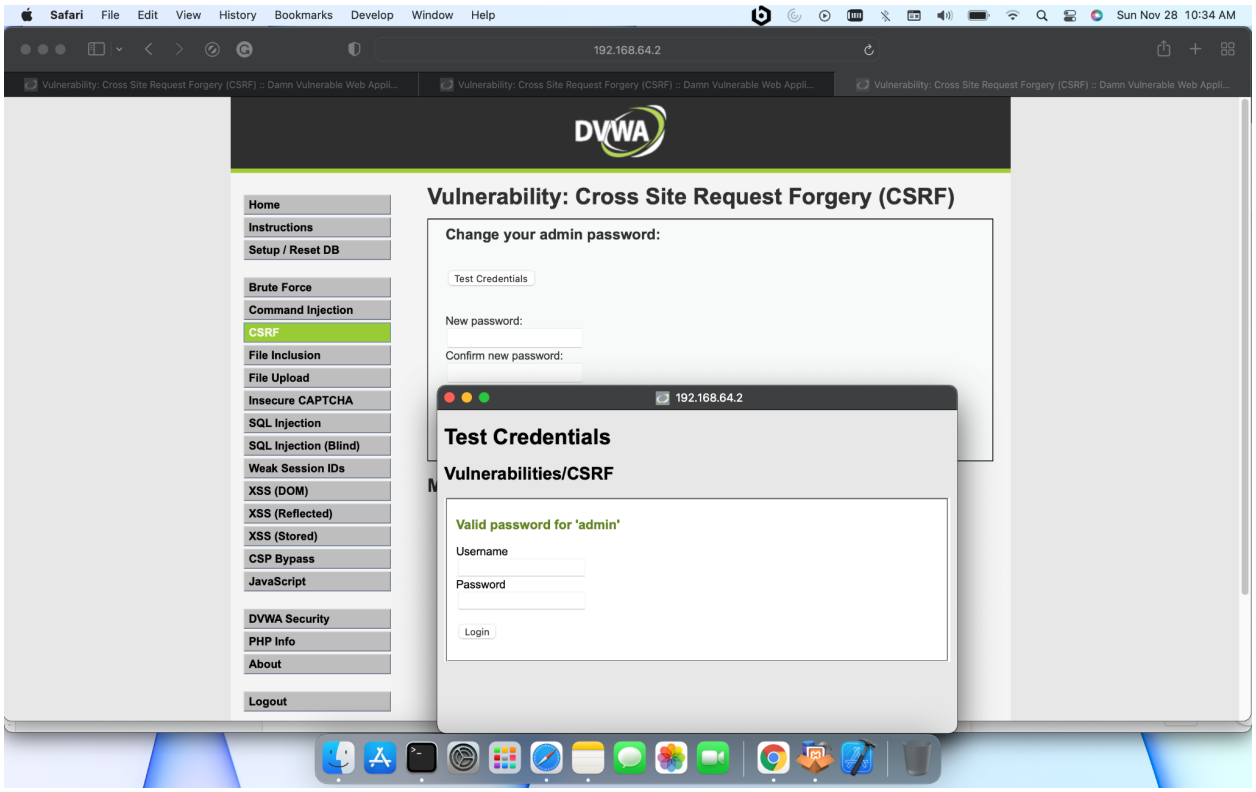
The function worked differently than normal use in that it allowed an unverified party to change the password of the user. This was done through the link which the user clicked on unknowingly.

The reason why this worked differently is because the origin of the link was not verified. Determining who generated the link is useful in preventing these kinds of attacks. Since a valid link was given, and no checking was done, this allowed the unverified party to change the password.

This vulnerability is really important because it can allow an adversary to change the password of any user that clicks on this link. This means that they will get access to their account fairly easily. Having access to the accounts of others could let them change more things about their profile and even use it to gain access to personal information.

To prevent these kinds of attacks one could use a CSRF token. This token is randomly generated by the site and helps the site verify that they were the ones that generated that URL. Therefore if an adversary tried to get a user to click on their malicious link, it would not change the password since there was no CSRF token, or it did not match what was expected.
https://portswigger.net/web-security/csrf





http://192.168.64.2/dvwa/vulnerabilities/csrf/?password_new=password&password_conf=password&Change=Change

The OWASP vulnerabilities that this relates to is insecure design and server-side request forgery. The first is because insecure design allows the adversary to generate a link that when clicked on will automatically change the users password. The second is because a URL was forged by the adversary to exploit this vulnerability.

Source: https://www.youtube.com/watch?v=Nfb9E8MJv6k

Vulnerability 4: File Inclusion

This function normally works by allowing users to read files that have been included on the site. Normally users can just click on the files that are listed there and their contents will be displayed to the user.

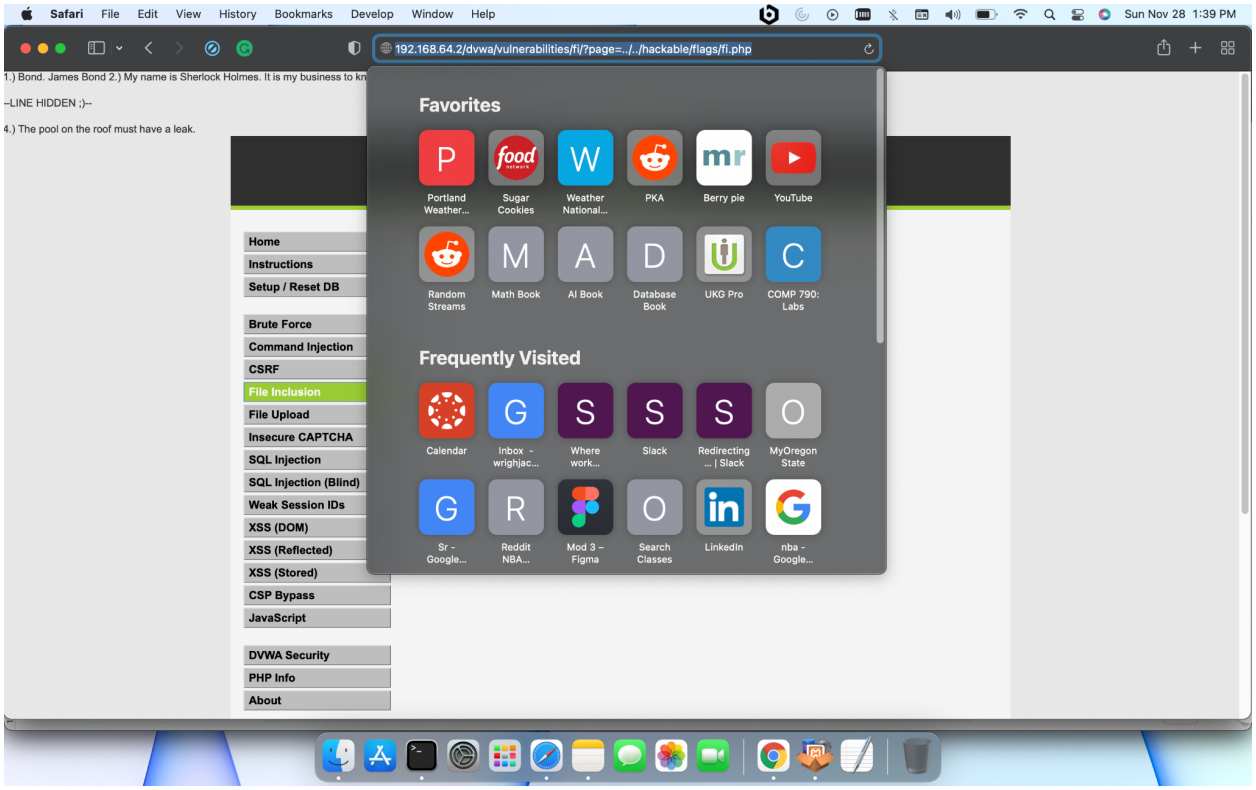
All it takes to exercise this vulnerability is the link of the website where these files are listed, as well as an understanding of where information is stored on the backend. To find this information, the command injection vulnerability can be used to snoop around the file structure. Once the destination of the file is known the contents can be printed to the screen using the link, and specifying the location at the end of the link.
(http://192.168.64.2/dvwa/vulnerabilities/fi/?page=../../hackable/flags/fi.php)

The feature worked differently than normal in that it brought up contents of files that were not listed on the page.

The reason why this works differently than normal is that the file that it brings up is identified through the url. Therefore, if the user specifies a file that is not currently listed on the page, it will still be brought up since the file exists in the given path. Therefore, this vulnerability works since the url is not checked to see if files in other parts of the system are trying to be accessed.

This is an important vulnerability because adversaries can use this to view contents of files throughout the server. If there were files containing password, or senstive information, they could be viewed using this method. Taking over peoples accounts, or stealing their information could be important losses that stem from this vulnerability. Similarly, this could allow them to execute malicious files, even ones that are on other servers.
https://www.neuralegion.com/blog/file-inclusion-vulnerabilities/

To prevent these kinds of attacks, input sanatation could be utilized to check for inputs in the URL. For example, parsing the URL and checking what parameter was passed in to the file variable would be important. Checking that file variable to make sure that it is only trying to access the small list of approved files could prevent attacks like these.
https://www.neuralegion.com/blog/file-inclusion-vulnerabilities/



This vulnerability falls under the Insecure design and Broken Access control sections of OWASP top 10. This is because the system is not designed secure enough since it allows any files to be accessed through the URL. Similarly, the access control is broken since other users who were not verified, such as myself, were able to access any file I wanted to.

Source: https://www.youtube.com/watch?v=KY58WcX7OZ4

Vulnerability 5: File Upload

This feature normally works by allowing users to upload files from their local device onto the web server. The user simply selects the file they want to upload, then presses the upload button.

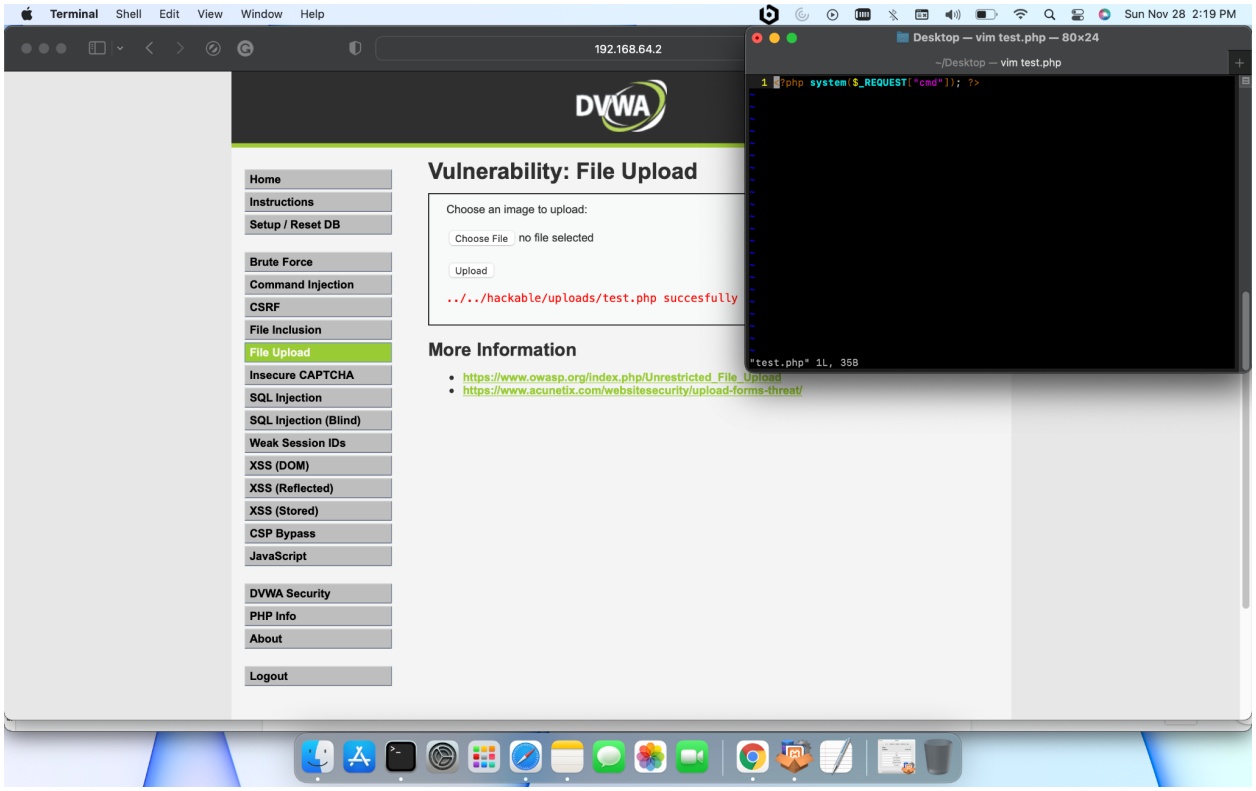
To exercise this vulnerability all you need is a malicious file. As mentioned above, you just upload this malicious file, and since the location is printed out after it is uploaded, all you need to do is go to that location and the file will be executed.

This feature doesn't actually work much differently than normal use. All that is happening is a file is being uploaded as it is intededed, however the contents of that file are not supposed to be allowed. In other words, the checking of the contents is the only thing that is missing causing this to behave differently than desired.

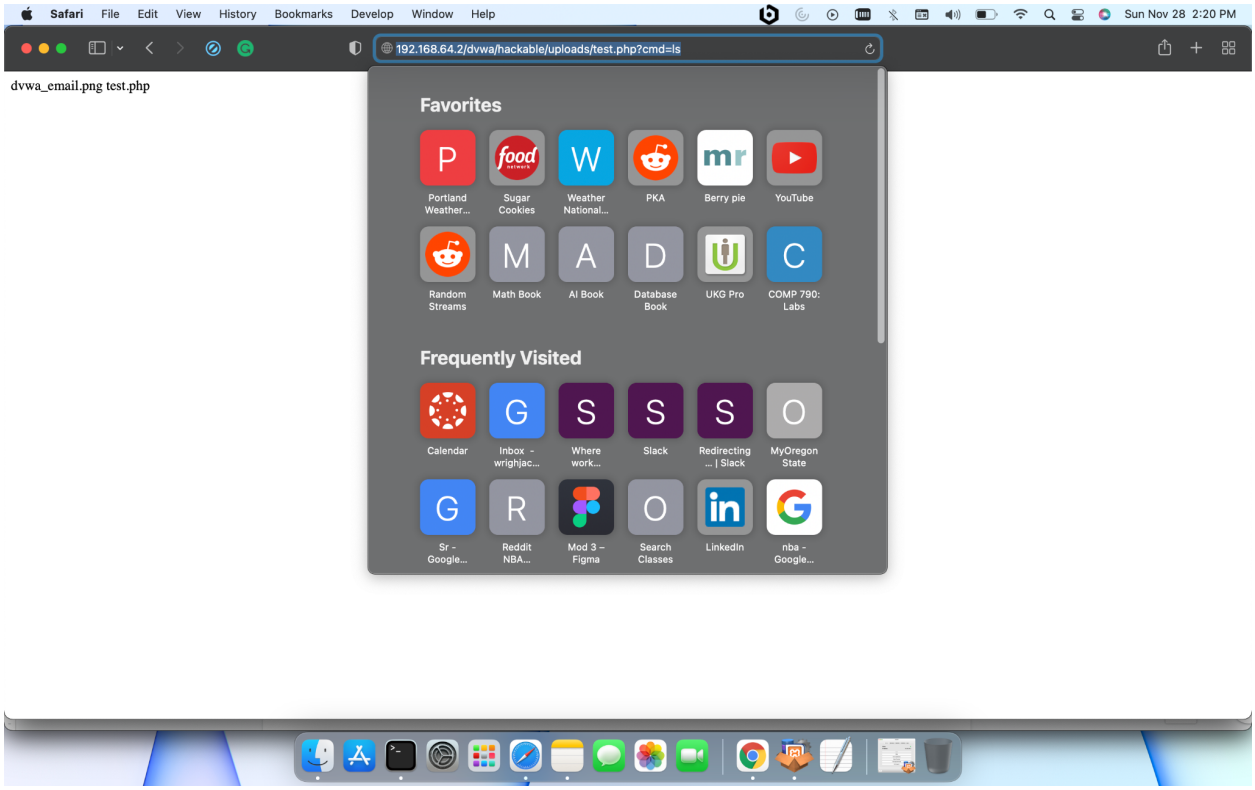
The reason why this worked differently is because the file content was never checked so all kinds of malicious files can be uploaded.

This is a really dangerous vulnerability, as it could allow an adversary to execute any kind of file or command on the web server. As mentioned in the tutorial video linked at the buttom of this section, a reverse shell could be created, giving the attacker control over parts of the server. Worst case, file upload could allow an adversary to gain complete control over the system
http://192.168.64.2/dvwa/vulnerabilities/upload/ - view help section

There are several different ways one could go about preventing this kind of attack. To start, the site could limit file uplaods to only a specific group of trusted users. Similarly, the site could make it so that only certain kind of files are allowed such as .txt files which could help limit the damage that they can do. Lastly, the site could try and check the content of the file, making sure no code is included in the file.
https://www.wordfence.com/learn/how-to-prevent-file-upload-vulnerabilities/



First created a php file which I uploaded to the site.



Next I navigated to the location of the new file, and updated a new parameter I included in that file.

The OWASP vulnerability that this relates to is Insecure Design. This is because the system was not designed in a way to check for certain content in files that are uploaded. In other words, the insecure design of the system allowed for malicious files to be uploaded.

Source: https://www.youtube.com/watch?v=K7XBQWAZdZ4

Vulnerability 6: SQL Injection

This feature normally works by having the user enter in a user ID which it uses to display the corresponding first and sur name.

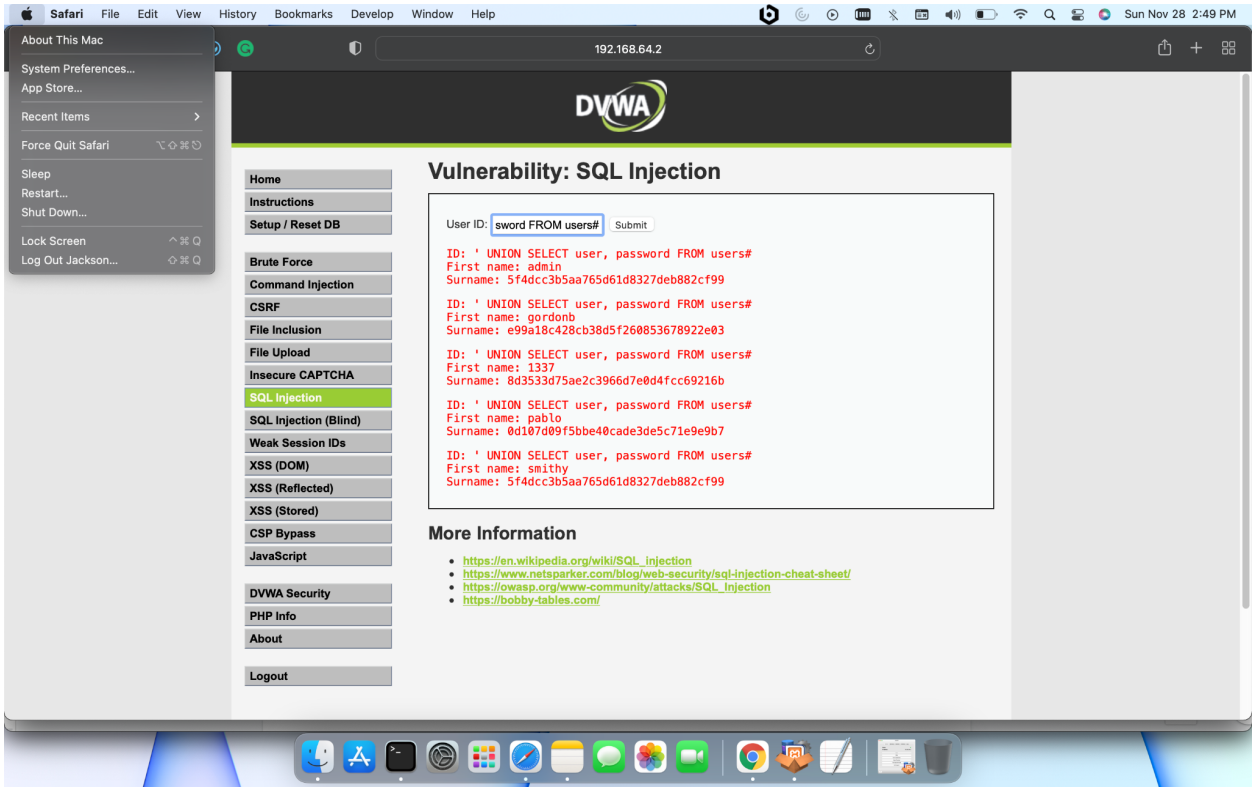
To exercise this vulnerability all you need is a SQL statement. First start off by including a single quote at the start of this statement. This will end the previous statement that it was trying to execute which in this case is searching for usernames with the matching user ID. After this include the statement that you want the system to also execute. In my case I used the statement ' UNION SELECT user, password FROM users# which selects all the username and passwords from the database and prints them out to the screen.

The feature worked differently than normal in that it printed out the usernames and passwords for all the users. Normally it is just supposed to print out the first and sur names. In this case it ran an SQL statement that was not intended to run.

The reason why this worked differently is because the input was not cleaned or checked by the system. This allowed the input to end the statement that was intended to run, and include a malicious statement. Similarly, having another statement that was not intended to be ran caused the system to behave differently.

This is another significant vulnerability that could allow the adversary to gain access to a lot of information within the database. In our case we were able to see the passwords and usernames of all of the users. We could also use this to delete entries in the database which would cause a lot of loss of information.

To fix this vulnerability you need to clean the input from the user. In this case since the user ID is all that is expected, you could clean the input to check for just numbers. Another way to clean the input would be to check for special SQL characters. In this case checking for SELECT and UNION keywords or the single quote could help prevent these kinds of attacks.



' UNION SELECT user, password FROM users#

The OWASP top 10 vulnerability that this relates to is injection. This is because users are able to inject SQL code that gets ran by the system.

Source: https://www.youtube.com/watch?v=5bj1pFmyyBA

Approach for Automating Tests

One of the main approaches I would take when trying to create an application for automating tests would be utilizing the wfuzz algorithm. The main reason for this is because it allows you to specify the cookies when accessing a URL. I used this algorithm for the brute force method and realized it could be used for many of the other vulnerabilities as well. This would work since it would change small sections of the URL which would allow for repeated attempts.

```
wfuzz -c -w ./realPass.txt -b 'PHPSESSID=40ajfqq31j6b1dcj8vg9n96lo2'
'http://192.168.64.2/dvwa/vulnerabilities/brute/?username=admin&password=FUZZ&Login=Logi
n'
```

Here the realPass.txt contains a list of passwords that I was checking for, and the -b specified the session I was currently in that was stored in the cookies. Finally the URL is included with FUZZ being the argument that is passwed in. For other vulnerabilities simply creating a list of different inputs you want and changing the URL for the specific attack would let you automate these kinds of tests.

For file inclusion vulnerability we could create a list of files we would like it to include then use the wfuzz algorithm to execute that link with a new file each time. This would automate the file inclusion vulnerability. For the SQL injection something similar would be done with a list of sql statements that gets iterated through and executed when included in the URL. The wfuzz algorithm tells you the word count of the return page after each execution. This can be used to determine when the site is acting differently and therefore would give you an idea of when there is a vulnerability.