

Damn Vulnerable Web Application Report

Theresa Bolaney

CS 370: Introduction to Security

Bram Lewis

December 1, 2021

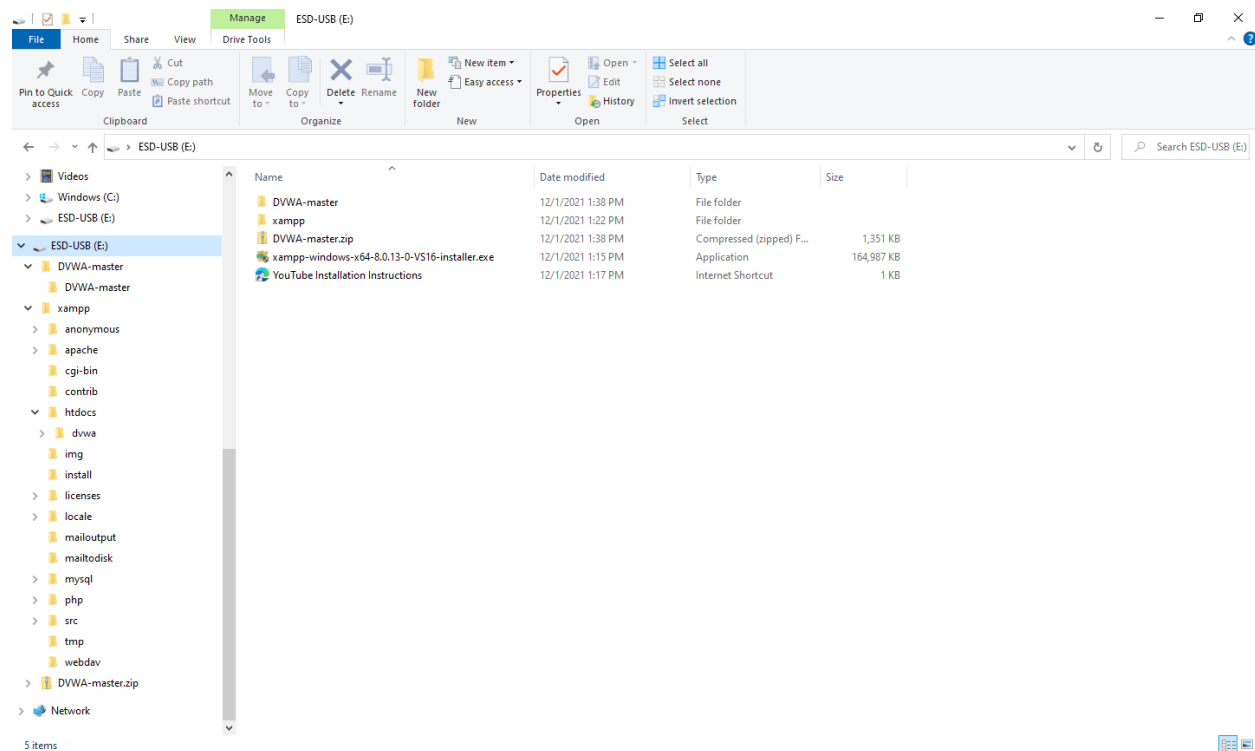
Setup

To set up DVWA, I followed video instructions provided by Ryan Dewhurst (Dewhurst, 2017).

This consisted of downloading XAMPP and DVWA onto a flash drive. I chose a flash drive as recommended by the professor, as this would help to ensure I did not leave DVWA running on accident. The XAMPP install consisted of Apache, MySQL, and PHP. I copied the DVWA files into the htdocs folder within XAMPP. After installation and changing some configuration settings (username and password), I was successfully able to run DVWA on localhost and create the database. Finally, I set the security level within DVWA to Low. A screenshot of my installation folder structure is below (Figure 1).

Figure 1

Setup Structure



Note: The flash drive (drive E) is shown. The major files are shown close on the middle/right. The left sidebar shows an expanded tree view of the folders, including the dvwa folder within xampp.

Vulnerabilities

My approach in tackling this assignment was to find at least one vulnerability across each of the major sections within DVWA (except CAPTCHA and File Upload). Each vulnerability is listed below with a number and given a descriptive title for ease of reading and navigating. The security level is set to Low for all entries unless otherwise noted.

1 – Brute Force: System Allows Unlimited Log-in Attempts

How feature normally works:

In a secure scenario, the system would stop the user from further log in attempts if they failed 3 (or another number) times or more.

How to exercise the vulnerability:

To exercise this, it is as simple as manually entering passwords over and over again. Or, one can use an automated script to do it for them.

How the feature works differently from normal use:

The system does not stop the user from trying an infinite number of login attempts.

Why this worked differently:

By viewing the source code, it's clear to see that the developer just created a simple if/else loop to check the given username and password against the database. There is nothing implemented to track the number of failed login attempts and stop further ones.

Why this is important:

If users have weak passwords, such as any that are listed in a dictionary of commonly used passwords, then it would be only a matter of time before a basic dictionary attack or brute force attempt would

break through. Since the system is also not tracking the number of attempts, this type of attack could go undetected until a bad actor actually does damage.

How to fix the vulnerability:

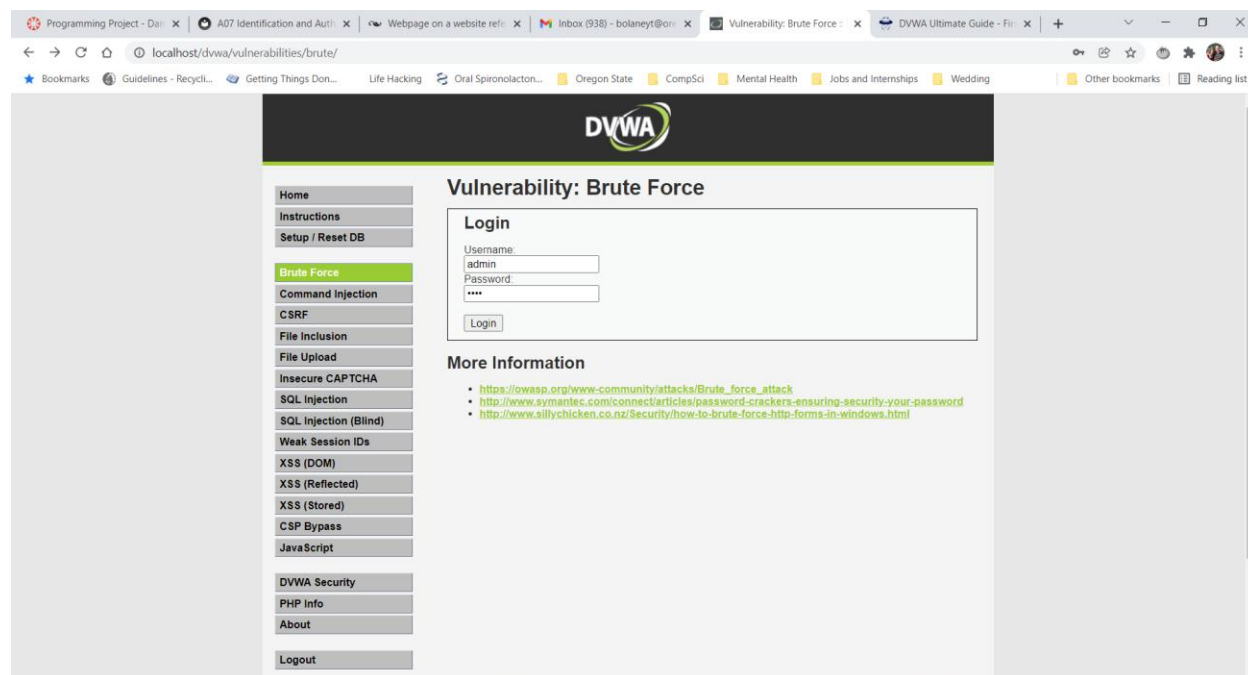
A way to track the number of login attempts per user id or session should be implemented. After a certain threshold, the user should be locked out for a certain amount of time or until an administrator resets their password. (OWASP Top 10 Team, 2021)

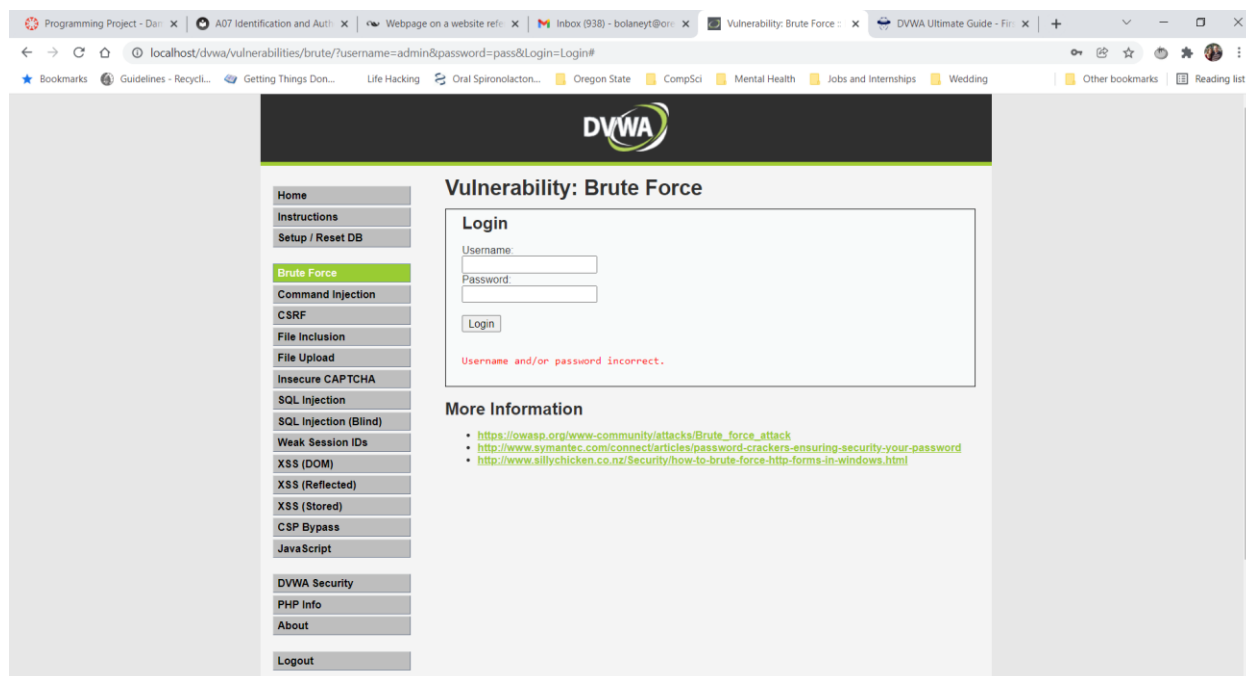
OWASP Top 10 vulnerability category:

A07 Identification and Authentication Failure

Steps to exercise vulnerability:

While this can be done with a variety of software tools, I elected for a manual approach. All this involves is manually typing new user names and passwords as long as I like. Below, I show that the error message and reaction from the website is the same as I attempt multiple password attempts for a user name.





Citation for exercising the vulnerability: (Giedrius, 2021, Brute Force section)

2 – Command Injection: IP Address Field Executes Commands

How feature normally works: The user can enter an IP address for a device and the website will attempt to ping it.

How to exercise the vulnerability: Instead of just entering an IP address, the user can follow it up with commands that could be executed on the operating system. In the example shown below, I added an escape character and then command “dir” after an IP address.

How the feature works differently from normal use: Normally, an entry field such as this should only accept an IP address, nothing else. This field accepts more than it should and executes it.

Why this worked differently: The field is not doing any input sanitation on the data it is being given.

According to the source code, the ping command is running whatever it is given directly at the command line. Ping will occur, but so will anything that follows.

Why this is important: If a bad actor were to do this, they could do essentially anything that an administrator at the command line could. This means granting themselves access, copying data, deleting data, etc.

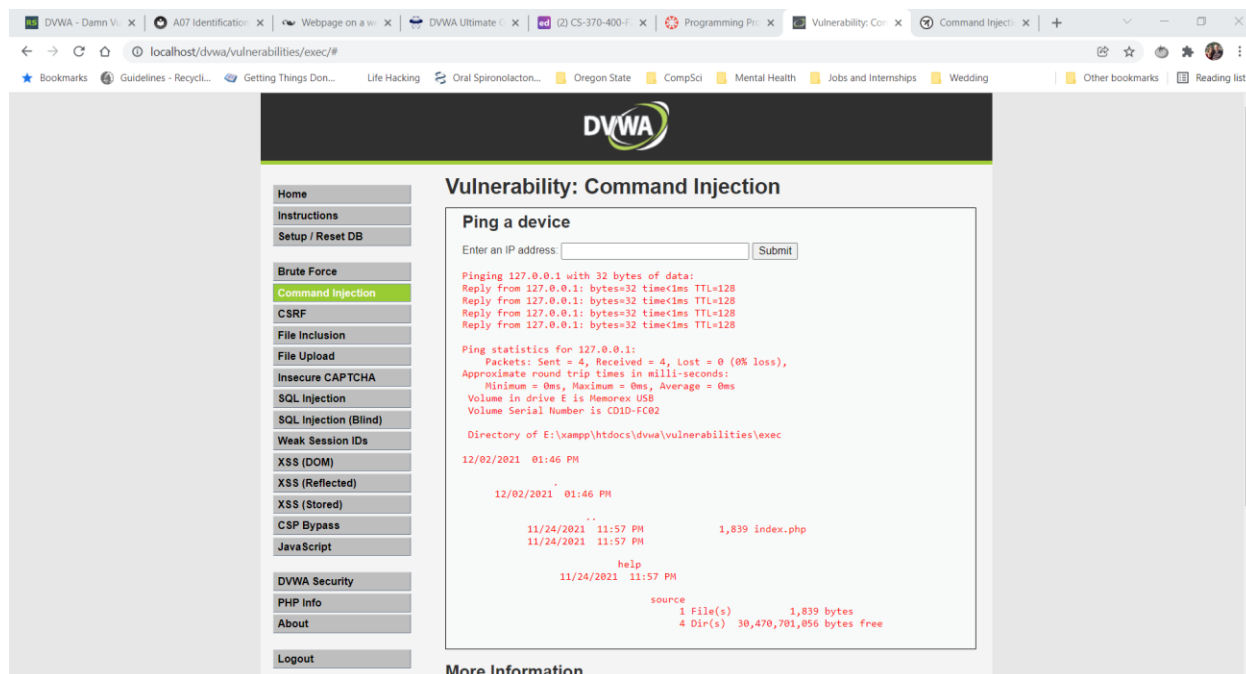
How to fix the vulnerability: Implementing input sanitation such that no shell escapes are allowed would prevent this from happening. (Zhong, n.d.)

OWASP Top 10 vulnerability category:

A03 Injection

Steps to exercise vulnerability:

Exercising the vulnerability is as simple as following an IP address with an escape character and a command. Below, I have entered "127.0.0.1 && dir" in the field.



Citation for exercising the vulnerability: (Giedrius, 2021, Command Injection section)

3 – Control Site Request Forgery: Create Link to Change Password

How feature normally works: A user who needs/wants to change a password can go to the appropriate page to change it. Otherwise, it does not change.

How to exercise the vulnerability: A bad actor can create a link that contains the appropriate fields for the HTML password elements, set the password to whatever they wish, then trick the end user into clicking on it. This will change the password without the user's consent.

How the feature works differently from normal use: Normally, any links that one clicks should not be able to initiate a password change. If one made the attempt, however, it should fail.

Why this worked differently: This vulnerability occurred because the HTML elements for the password are visible in the URL and can be easily manipulated. All the current code does is verify if the currently

provided passwords match each other and, if they do, it updates the database with the new password. It pulls in the current user associated with the session automatically and does not do any more checks. This effectively means that if a user is already logged in when they click the bad link, the hacker's attempts will succeed.

Why this is important: Changing a user's password is an obvious impact. If that user happens to be an administrator, the bad actor now has access to the entire system and can do anything that this administrator was able to do.

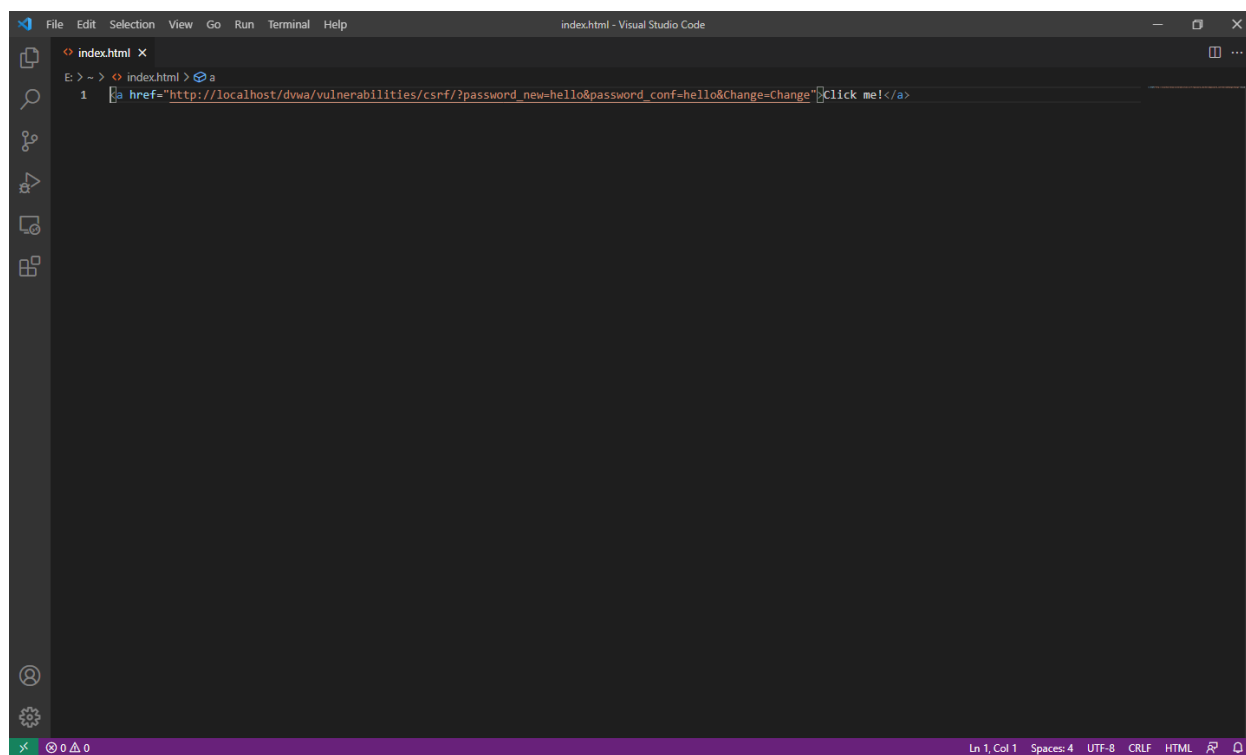
How to fix the vulnerability: One suggestion is to check the referrer header in the client's HTTP request. By ensuring that the HTTP request comes from the original site (and not the bad actor's fishy site) means that other sites will not function. (KirstenS, Cross Site Request Forgery (CSRF), n.d.)

OWASP Top 10 vulnerability category:

A01 Broken Access Control, A07 Identification and Authentication Failures

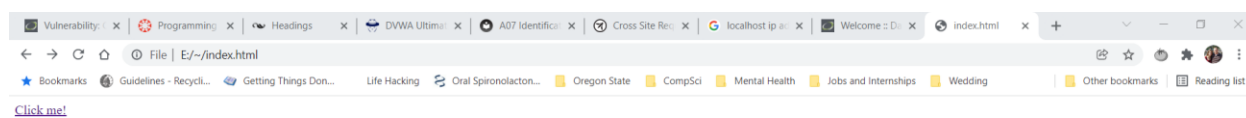
Steps to exercise vulnerability:

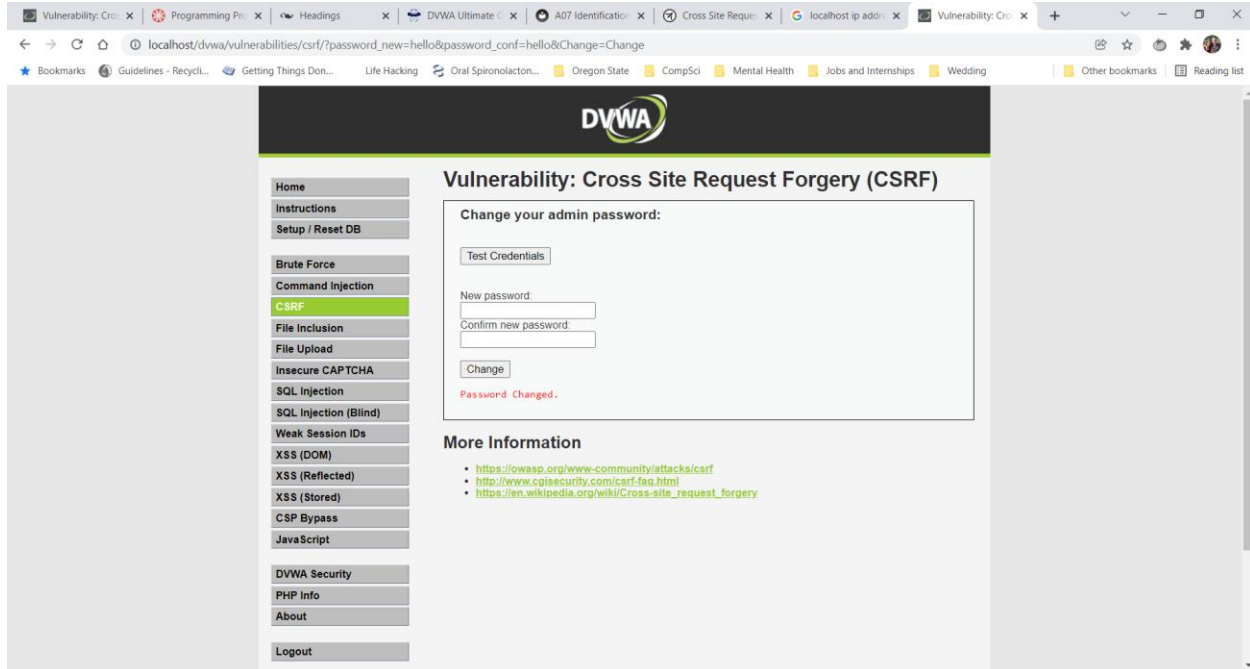
To expose this, create a simple HTML file that contains a link which leads to a modified URL header filled with the bad actor's chosen password. Clicking on this link while logged in as the administrator (this must occur in the same session/browser) will bypass the normal log in screen and immediately set the user's password without consent.



```
index.html x
E:\> ~ > index.html > a
1  <a href="http://localhost/dvwa/vulnerabilities/csrf/?password_new=hello&password_conf=hello&Change-Change">Click me!</a>
```

Ln 1, Col 1 Spaces: 4 UTF-8 CRLF HTML





Citation for exercising the vulnerability: (Giedrius, 2021, CSRF section)

4 – File Inclusion: Navigate to Other Files Using URL

How feature normally works:

When a user accesses a specific file on the server (such as by clicking a link for it), they should have access only to that file.

How to exercise the vulnerability:

The URL can be changed to access a different file on the server (local file inclusion) or elsewhere (remote file inclusion).

How the feature works differently from normal use:

A user is able to gain access to files they normally should not have access to.

Why this worked differently:

In this case, the program does not sanitize the inputs the user enters into the URL.

Why this is important:

Imagine a bad actor who is able, perhaps through another exploit, to upload a malicious executable or other file onto the local server. They would then be able to access and run that file by merely altering the URL.

How to fix the vulnerability:

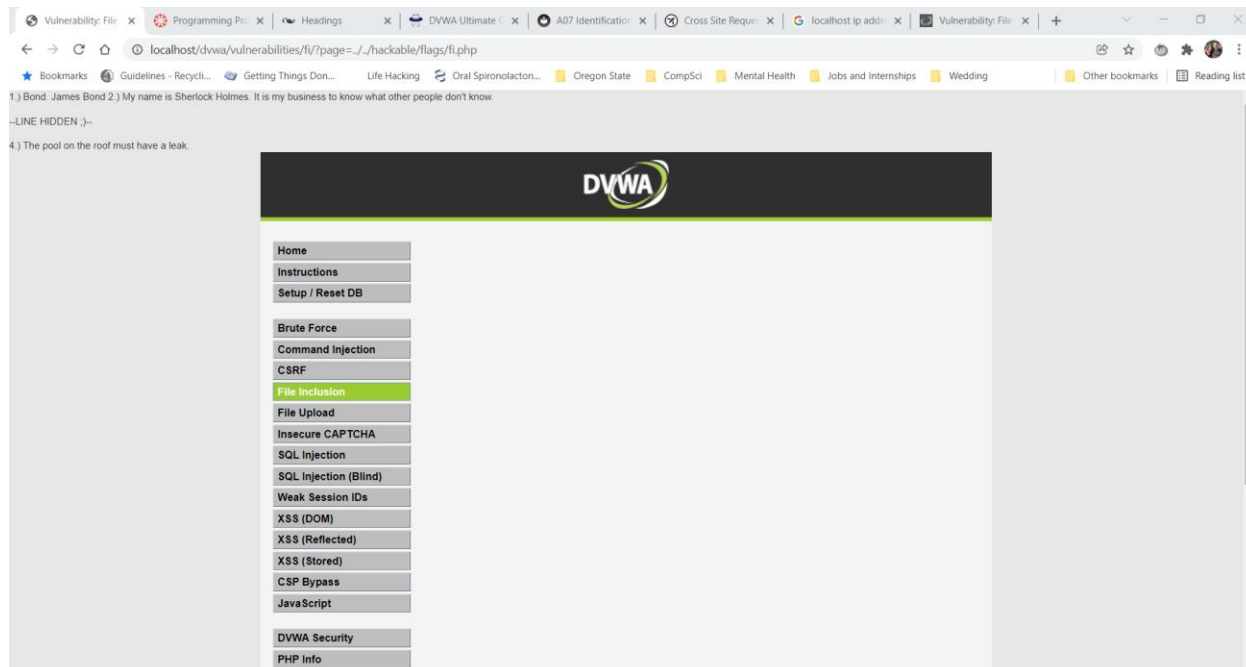
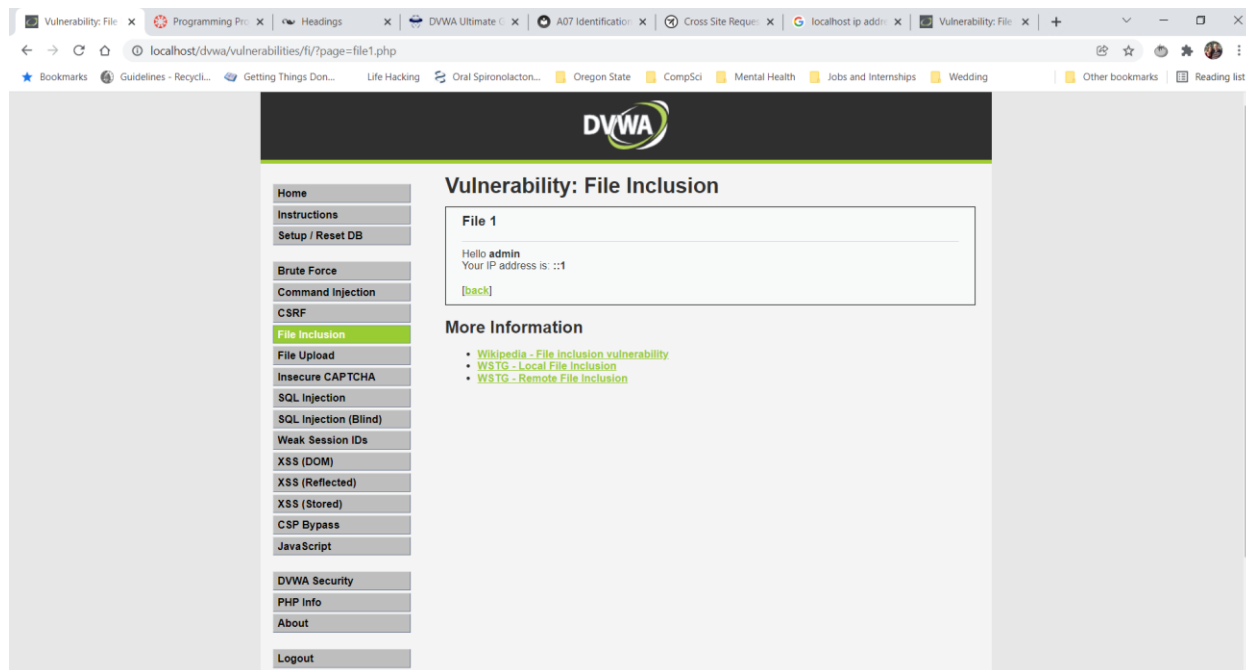
Implement a way to avoid passing user-submitted input to any filesystem/framework API. (Testing for Local File Inclusion, n.d.)

OWASP Top 10 vulnerability category:

A01 Broken Access Control

Steps to exercise vulnerability:

Exercising this vulnerability can be done by changing the URL to direct to another page/document. I start by clicking on the file 1 link. Then, I replace it in the header with the file path for the provided file “fi.php” mentioned in the help section. This directs me to a new HTML page with several quotes.



Citation for exercising the vulnerability: (Giedrius, 2021, File Inclusion section)

5 – SQL Injection: Obtaining User Information

How feature normally works:

This field, presumably for an administrator, allows the user to enter the ID of another user. That second user's password is then pulled from the SQL database and shown on the screen.

How to exercise the vulnerability:

One can display this vulnerability by entering an escape character (in this case ') and a line of SQL code.

How the feature works differently from normal use:

Under normal circumstances, the field should only accept a numeric value for the input. Any additional text should be ignored, not performed directly on the database.

Why this worked differently:

The input is not sanitized. Additionally, raw SQL commands are sent to the database.

Why this is important:

This could cause devastating effects such as the theft of large amounts of data, or even the destruction of that data. A user could do effectively anything they wanted with the database. They could sneak in and read sensitive information, change it however they want, or delete it at will.

How to fix the vulnerability:

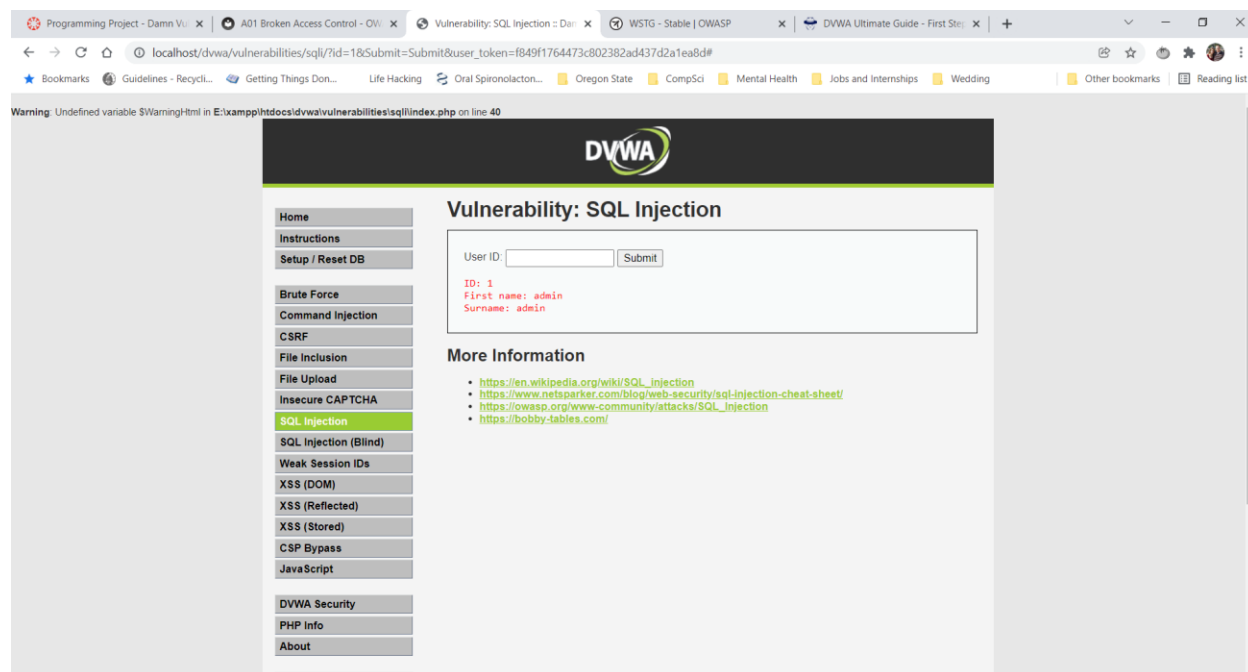
Since this field only requires a numeric input, I would sanitize that and reject any inputs that were not allowed. This is especially true of the escape characters, as these characters are what enables this issue to occur. (kingthorin, n.d.)

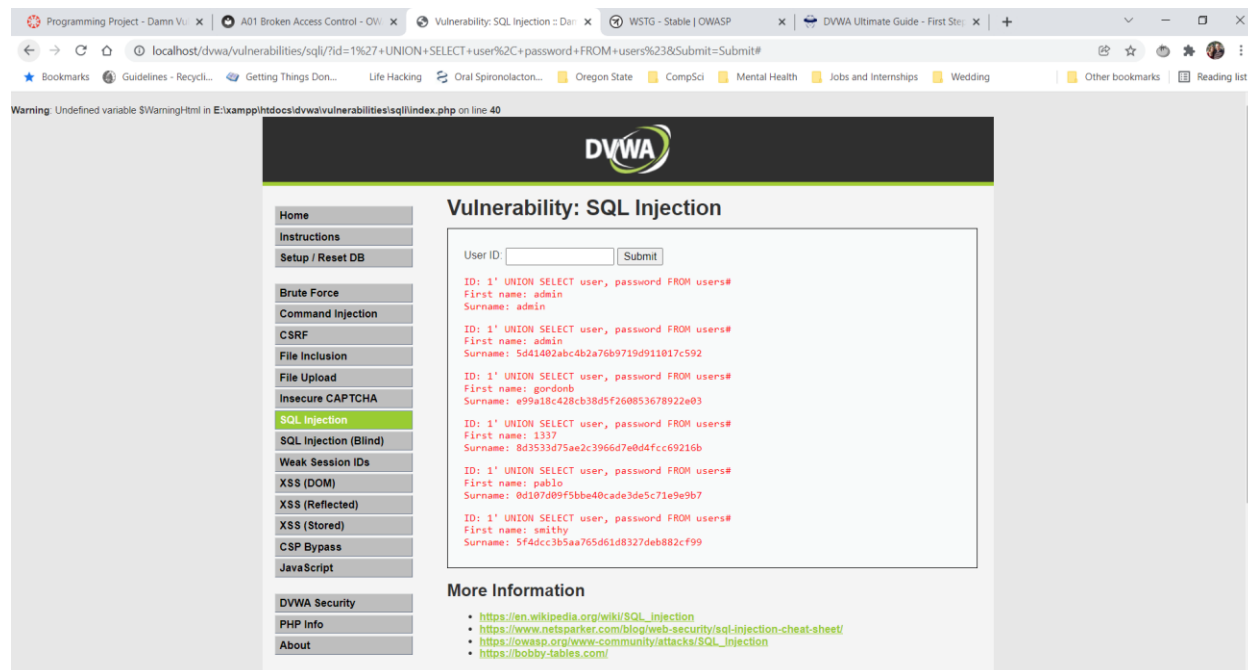
OWASP Top 10 vulnerability category:

A01 Broken Access Control, A03 Injection

Steps to exercise vulnerability:

The first screenshot below shows how the feature should work. If I enter “1”, the user name and password for the administrator is returned. The second screenshot shows what happens when I enter the valid ID of “1” but follow it with an escape character and a line of SQL code requesting all usernames: 1' UNION SELECT user, password FROM users#. Thankfully, the password is hashed so it is not displayed.





Citation for exercising the vulnerability: (Giedrius, 2021, SQL Injection section)

6 – Weak Session IDs: IDs are Predictable

How feature normally works:

The Generate button will set up a new cookie called dvwaSession whenever it is clicked.

How to exercise the vulnerability:

The vulnerability is inherent and occurs automatically whenever the button is clicked. It can be seen by viewing the cookie directly under the Application/Storage tab of the developer tools' window.

How the feature works differently from normal use:

Any time a cookie is generated, it should have a unique ID. While that is technically true here, the problem lies in the fact that it can be easily predicted. Every time the button is clicked, the cookie value increments by 1.

Why this worked differently:

Looking at the source code, this is occurring simply because the `last_session_id` value is being incremented and then that is populating the `$cookie_value`.

Why this is important:

A session ID this simple is easy to guess or calculate. Doing this, a malicious user could have an easy way to get into the system without needing to brute force a password. Once inside, they could then do damage such as accessing and stealing vulnerable data, all depending on whatever authorizations were enabled for the true user of that original session.

How to fix the vulnerability:

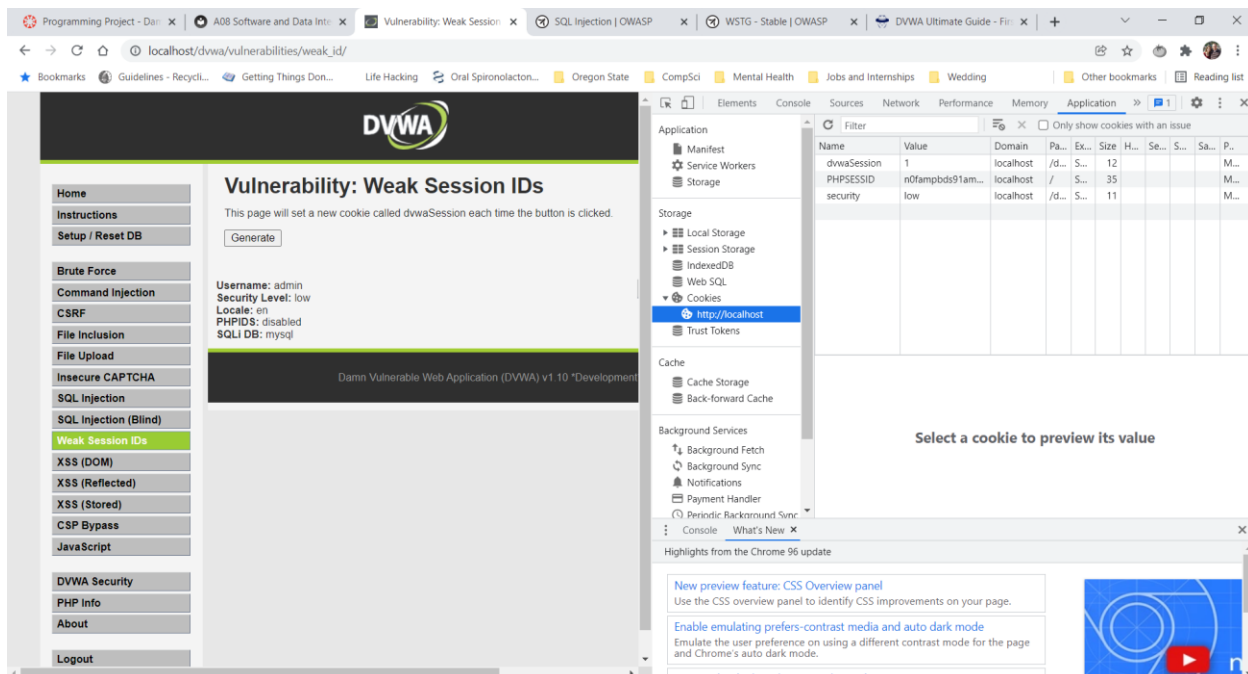
Creating a randomly generated session ID rather than one so predictable would prevent a bad actor from being able to guess an ID. (Session Management Cheat Sheet, n.d.)

OWASP Top 10 vulnerability category:

A07 Identification and Authentication Failures

Steps to exercise vulnerability:

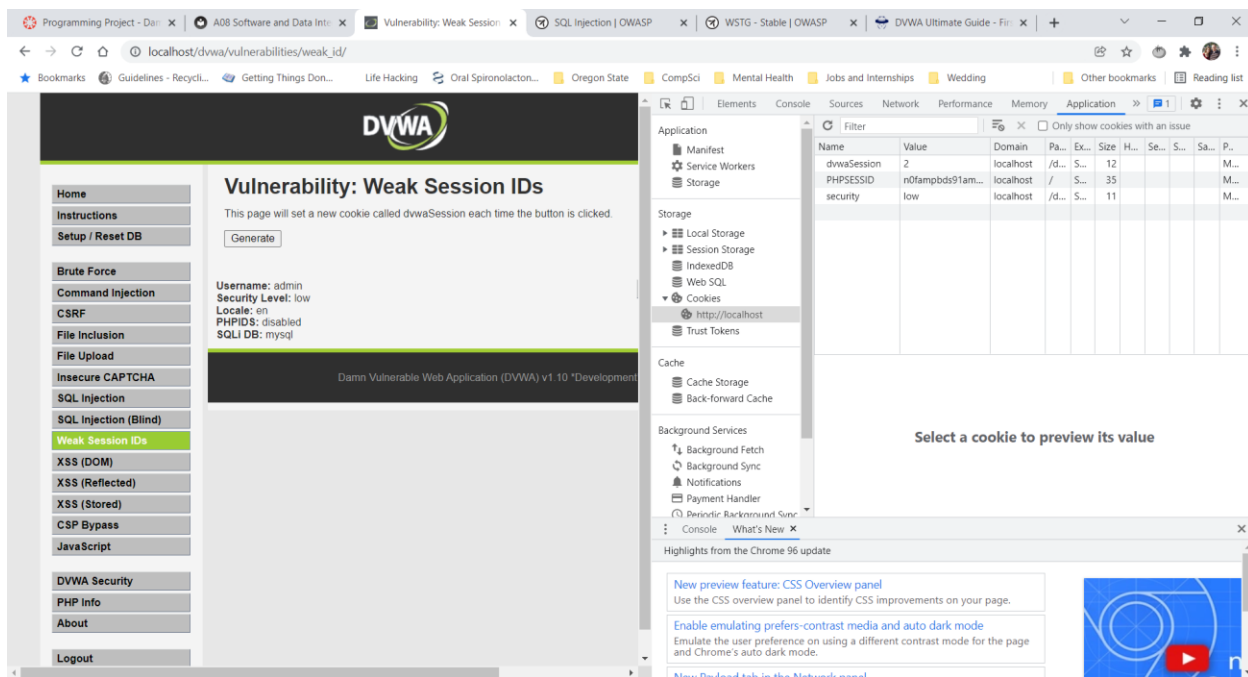
This vulnerability occurs automatically when the button is pressed. Pressing it once and viewing the cookie shows that it is set to 1. Doing so again sets it to 2.



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface at the URL `localhost/dvwa/vulnerabilities/weak_id/`. The page title is "Vulnerability: Weak Session IDs". It includes a sidebar with navigation links and a main content area with a "Generate" button and user information: Username: admin, Security Level: low, Locale: en, PHPIDS: disabled, SQLI DB: mysql. The application version is v1.10 *Development*.

The Chrome DevTools Application tab is open, displaying a table of cookies:

Name	Value	Domain	Path	Expires	Size	HttpOnly	Secure	SameSite	Priority
dwSession	1	localhost	/	S...	12				M...
PHPSESSID	n0fampbds91am...	localhost	/	S...	35				M...
security	low	localhost	/	S...	11				M...



This screenshot is identical to the one above, showing the DVWA interface and the Chrome DevTools Application tab with the same cookie table.

Citation for exercising the vulnerability: (Giedrius, 2021, Weak Session IDs section)

7 – XSS (DOM): Scripts Executed in URL

How feature normally works:

This field is a simple drop-down menu. A user can click on it to select their preferred default language.

How to exercise the vulnerability:

The URL can be changed by the user such that a script is appended and runs.

How the feature works differently from normal use:

Normally, if a user would try to enter a script at the end of the URL, this would be ignored and the user would be directed to an error page.

Why this worked differently:

Looking at the source php, there are quite literally no protections in place to prevent this. This means the input is not checked at all.

Why this is important:

A bad actor could run a malicious script that could steal cookies, session tokens, or other data.

Additionally, the HTML content of the page itself could be changed.

How to fix the vulnerability:

All input should be sanitized! Anything that indicates JavaScript should be cleaned out, and the user should be directed to an error page. (KirstenS, Cross Site Scripting (XSS), n.d.)

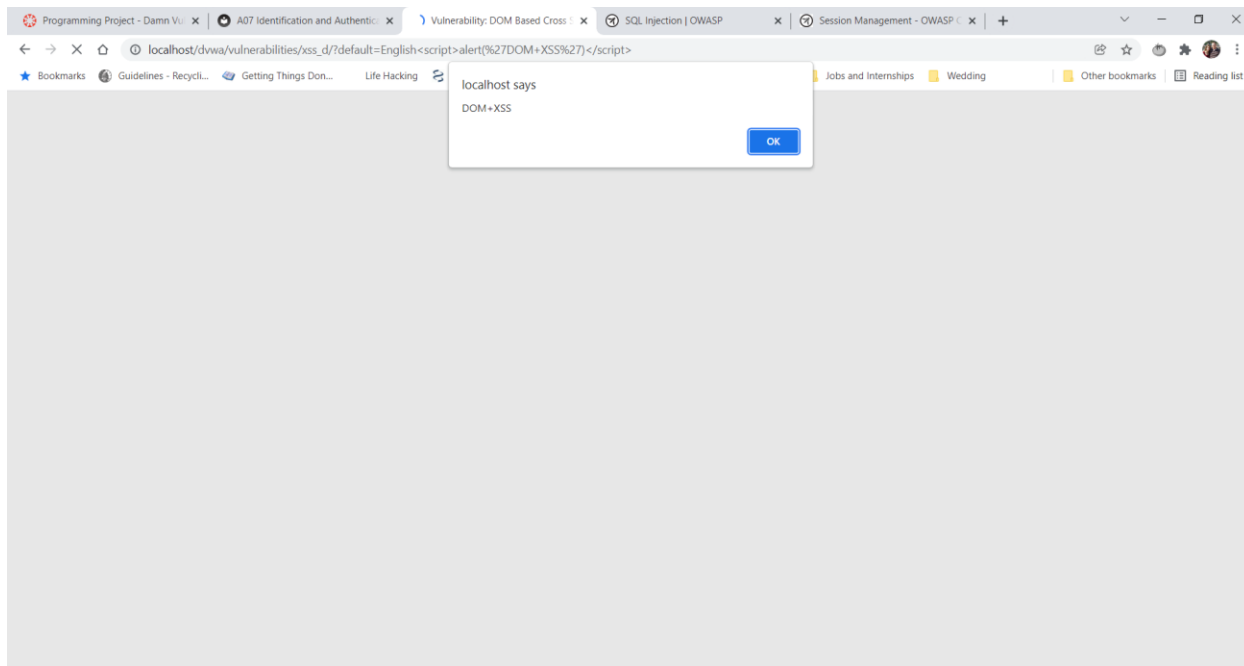
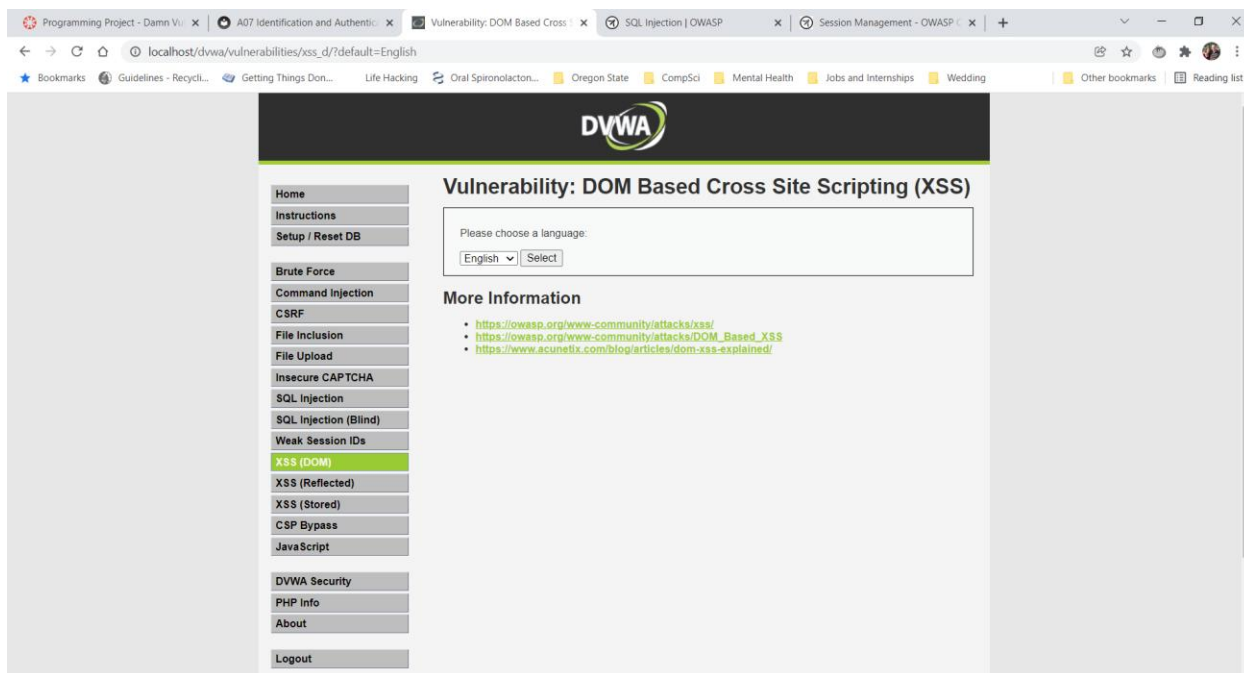
OWASP Top 10 vulnerability category:

A03 Injection

Steps to exercise vulnerability:

The first screenshot shows what the URL changes to once English is selected in the dropdown menu.

After that, it is easy to append custom text (in this case a script to cause a popup) to the end of the URL, and it will be accepted.



Citation for exercising the vulnerability: (Giedrius, 2021, XSS (DOM) section)

Summary

Over the course of testing the DVWA, I have reproduced at least 8 different vulnerabilities within the site. I acknowledge that there are more bugs out there, and likely will be even more found in the future as new tools are developed and new holes are exploited. Testing is critical, and this has given me some ideas for how I might approach automating known tests for security purposes.

To create an algorithm, I started by assessing the OWASP website. That site has both the list of top ten vulnerabilities as well as cheat sheets and other guides for how to both test and prevent these issues. It became immediately apparent that input validation/sanitation was a very common theme among many of these issues. Automation would be tricky, but using a tool such as Sikuli could make things easier. Sikuli uses image recognition to identify GUI components, and after setting it up with an ideal test (passing cases), it would be able to report if any test cases did not match the visual results of the passing case. (Hocke, n.d.)

Below, I have posted a possible algorithm via bullet points. The intent here is to evaluate all areas where a user can interact with the webpage and run the tests against those areas. Every area begins first with input sanitation as having that in place would prevent many problems with injections of various sorts.

- Start by taking a snapshot of the session (cookies, URL, user information, etc)
- Identify an interactive piece of the web page
 - Create passing image screenshots in Sikuli of how this feature should work under normal conditions
 - Search for the next interactive piece, repeat until there are no more
- Once we know what the positive cases look like, we loop through the following tests for each area

- Input sanitation: no escape characters allowed, and only integers for numeric fields, etc
- URL navigation: if a malicious URL is entered, it should be rejected
- SQL injection: likely covered by input sanitation above, but test it against just in case
- JavaScript injection: same as above
- Command injection: same as above
- Cookies: compare against first snapshot to make sure there are no major changes
- Password fields: attempt to add common passwords that would be caught in a Bloom filter

References

- Dewhurst, R. (2017, May 22). *Installing Damn Vulnerable Web Applications (DVWA) on Windows 10*. Retrieved from YouTube: <https://www.youtube.com/watch?v=cak2lQvBRAo>
- Giedrius. (2021, April 7). *DVWA Ultimate Guide - First Steps and Walkthrough*. Retrieved from Bughacking: <https://bughacking.com/dvwa-ultimate-guide-first-steps-and-walkthrough/>
- Hocke, R. (n.d.). *SikuliX by Raiman*. Retrieved from SikuliX: <http://sikulix.com/>
- kingthorin. (n.d.). *SQL Injection*. Retrieved from OWASP: https://owasp.org/www-community/attacks/SQL_Injection
- KirstenS. (n.d.). *Cross Site Request Forgery (CSRF)*. Retrieved from OWASP: <https://owasp.org/www-community/attacks/csrf>
- KirstenS. (n.d.). *Cross Site Scripting (XSS)*. Retrieved from OWASP: <https://owasp.org/www-community/attacks/xss/>
- OWASP Top 10 Team. (2021). *A07:2021 - Identification and Authentication Failures*. Retrieved from OWASP: https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/#how-to-prevent
- Session Management Cheat Sheet*. (n.d.). Retrieved from OWASP: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html
- Testing for Local File Inclusion*. (n.d.). Retrieved from OWASP: https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/07-Input_Validation_Testing/11.1-Testing_for_Local_File_Inclusion
- Zhong, W. (n.d.). *Command Injection*. Retrieved from OWASP: https://owasp.org/www-community/attacks/Command_Injection