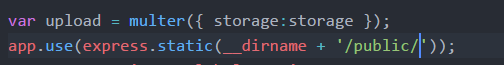
**Assignment 1b:**

**Write up of Vulnerabilities found in my Application**

1. **Image File Name doesn’t work if it has whitespaces.**

The file that I have been uploading till now had no whitespaces in them. If I uploaded one file with a space then, I get the success upload message, but the image is not visible when I check the module where all images are stored. I conclude that my application is not able to handle whitespaces in the file name format.

To resolve this and upload a file path that has spaces, we must introduce a backslash character in the path. In my code, I have escaped the file name by using “\”. We need to escape the backslash character so that it makes it in the output string.



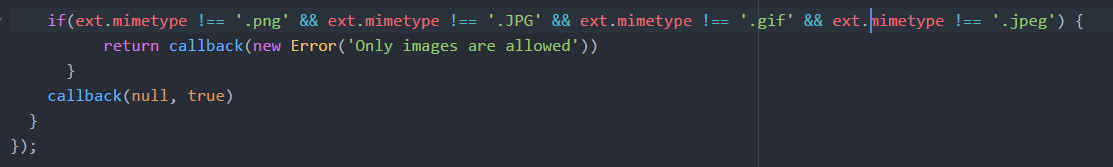
1. **Limiting the application to only upload Image formats**

A malicious user might use a .bin extension file or a bootstrap file with a script that might trigger the application to fail at certain points. This is possible because the user can upload file formats with any extension and its not limited to image types.

We can limit the file types so that the user can upload only certain types of file formats which are limited to only image formats (png , jpeg).

I have used mimetype which checks the extension of file and allows to upload those formats which are not restricted in the application.

This would help to avoid any misuse with the system.



1. **NoSQL Injection Test**

The attack tries to inject code when the inputs are not sanitized, and the solution is simply to sanitize them before using.

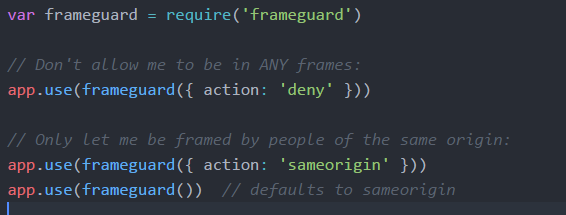
db.getCollection('images').find({'createdAt':{$gte:""}})  
Therefore while inserting the caption the user can have {$gte:””} (which says that greater than null) all the records can be retrieved for a given date.

If you are using Mongoose, you don't need to sanitize the inputs. In this case, you just need to set the properties to be typed as string. If someone passes an object like { $ne: null }, Mongoose will convert it to a string and no harm will be done.

I tried using most common no sql injection from this <https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/NoSQL%20injection/Intruders/NoSQL.txt> to test the application and could not find any vulnerability.

1. **ClickJacking**

Clickjacking is an attack that tricks a web user into clicking a button, a link or a picture, etc. that the web user didn’t intend to click, typically by overlaying the web page with an iframe. This malicious technique can potentially expose confidential information or, less commonly, take control of the user’s computer. For example, on Facebook, a clickjack can lead to an unauthorized user spamming your entire network of friends from your account.



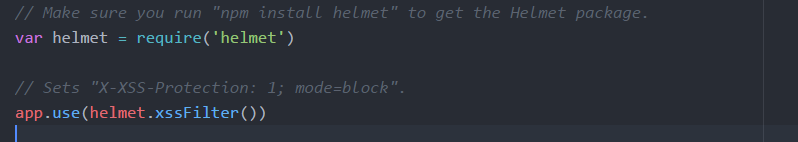
FrameGuard has three modes: DENY, SAMEORIGIN, and ALLOW-FROM, defaulting to SAMEORIGIN. If your app does not need to be framed (and most don't) you can use DENY. If your site can be in frames from the same origin, you can set it to SAMEORIGIN. If you want to allow it from a specific URL, you can allow that with ALLOW-FROM and a URL.

1. **X-Powered Header**

XSS, the most prevalent web application security flaw, occurs when an application includes user supplied data in a page sent to the browser without properly validating, escaping, or sanitizing that content. An XSS attack occurs when an attacker sends text-based scripts that exploit the interpreter in the browser.

The application can be exploited by the code below-  
<script>eval(‘alert(“your system is hacked”);’)</script>

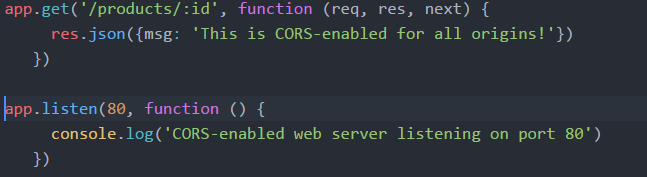
Helmet’s xssFilter is a relatively simple middleware that will set the X-XSS-Protection header. It is a popular library to prevent XSS and SQL injection.



1. **Cross-Origin Resource Sharing (CORS)**

The API initially accepted requests from anyone as the Access-Control-Allow-Origin allowed all. This could have made the application open to everyone and they could have access to server response. So any website could make request on your behalf and could target your server. This is a security risk so by adding this snippet it will be open only to local host.





**Partner Web Application Vulnerabilities: (Akshay Hazare)**

1. **Requirement of Password:** When I try to login and create a new account, I am not asked for password requirements like there should be a special character or CAPS character. By using Rainbow Crack (a popular tool for password tries and brute force attack) it is possible to break in after a probability of 1 in million tries.
2. **ClickJacking:** Django contains clickjacking protection in the form of the X-Frame-Options middleware which in a supporting browser can prevent a site from being rendered inside a frame. It is possible to disable the protection on a per view basis or to configure the exact header value sent.
3. **NoSQL Injection:** Akshay’s application has querysets are protected from SQL injection since their queries are constructed using query parameterization. A query’s SQL code is defined separately from the query’s parameters. Since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver.
4. **XSS Scripting:** This is possible in the application. But can be disabled if the auto escaping feature is turned off.
5. **Host Header Validation:** Web server configurations are susceptible to fake Host headers, Django validates Host headers against the ALLOWED\_HOSTS setting in the django.http.HttpRequest.get\_host() method.  
   A fake Host value can be used for Cross-Site Request Forgery, cache poisoning attacks, and poisoning links in emails.