Encrypted Signal – An Encrypted File Sharing System

Encrypted Signal is a full stack web application where I have practiced different secure programming aspects. The main purpose of this application is to share an encrypted file with another registered user. The file will be encrypted at the backend with user defined password in AES encryption.

Since the file can only be shared/uploaded by logged in user, the first UI is the sign up/login page. At this moment, there's no password recovery option, which will be added later.

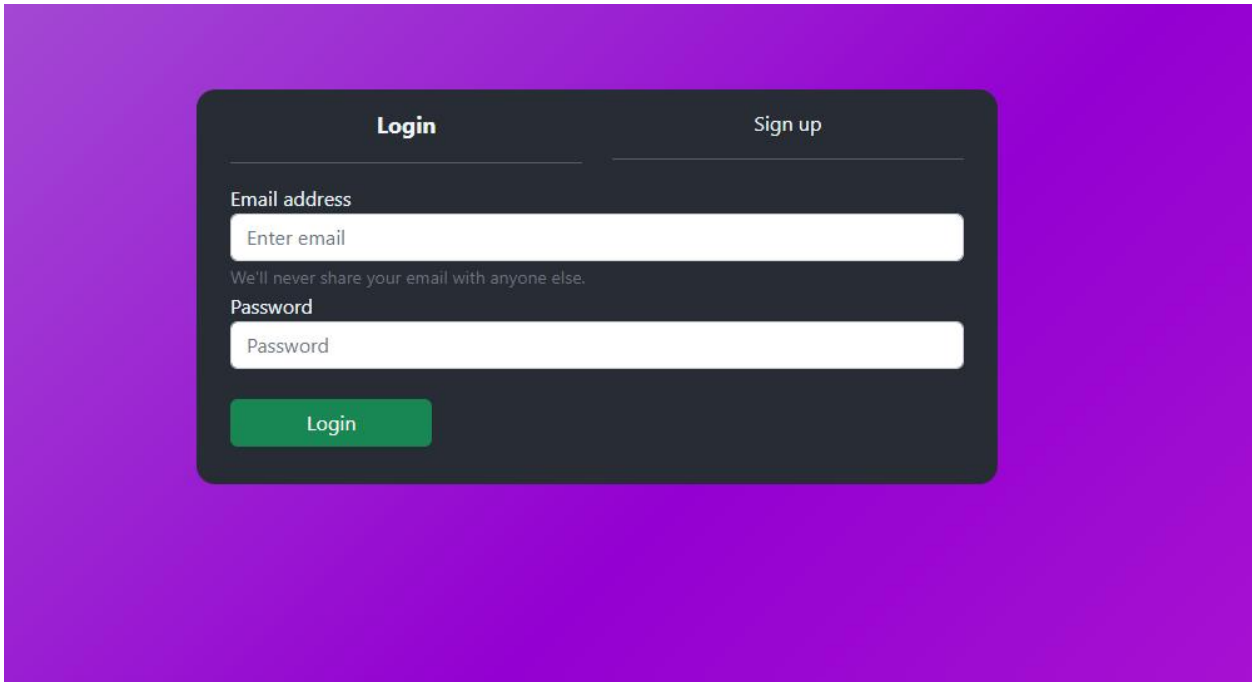


Figure 1: Login/Sign Up page

However, if a user is already logged in (based on session ID), then they will be redirected to the main dashboard. If we look at figure 2, we can see the figure marked with 1,2,3 and 4 section. Section 1 is the main Encryption and Upload section. If a user selects a file to upload, more options will appear. Section 2 is the list of files. Each user will get 3 options from decrypt & download, share, and remove to select and if the file is only shared with them, then they will only get to decrypt & download the file. Other options won't be visible to them. Section 3 is the choice between Your Files and Shared Files. Depending on which section is enabled, the file list will load accordingly. The final section, section 4 is the number of files currently on that particular list.

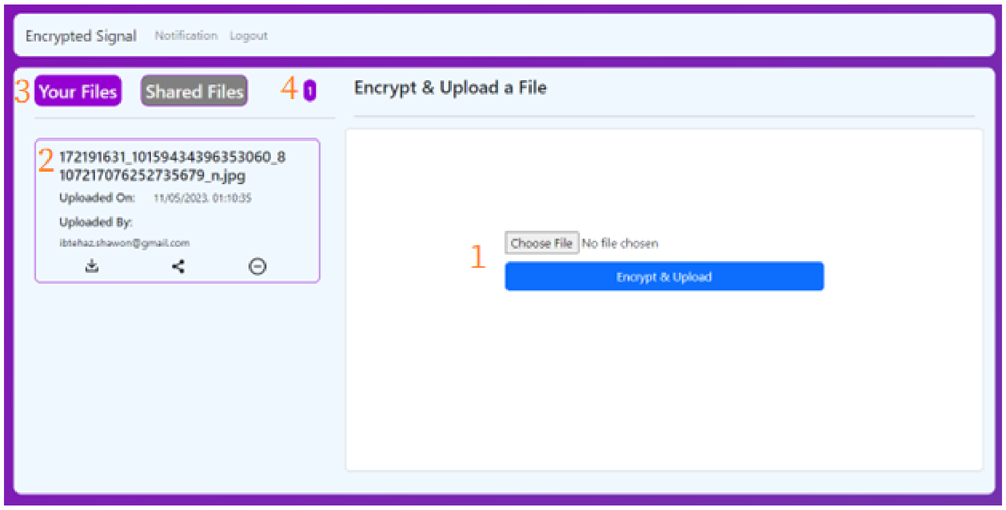
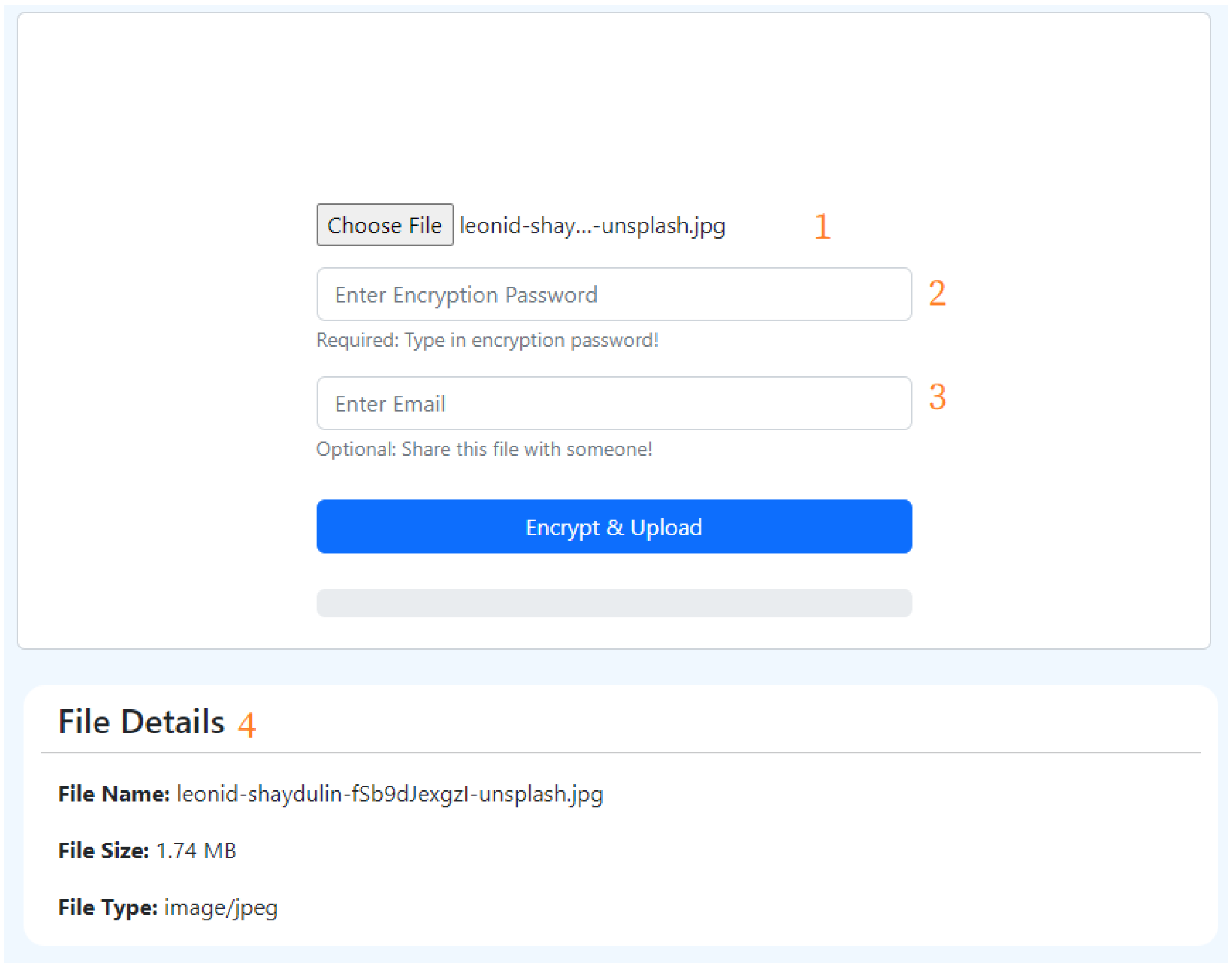


Figure 2: Primary Dashboard

Now, after a user select a file to upload (section 1), it will immediately show a new input field to give the encryption password (section 2), another optional field to enter another registered email address to share the file with (section 3) and finally some file details (section 4). All the information from section 4 will be uploaded to the server.


Choose File
leonid-shay...-unsplash.jpg
1
Enter Encryption Password
2
Required: Type in encryption password!
Enter Email
3
Optional: Share this file with someone!
Encrypt & Upload
File Details
4

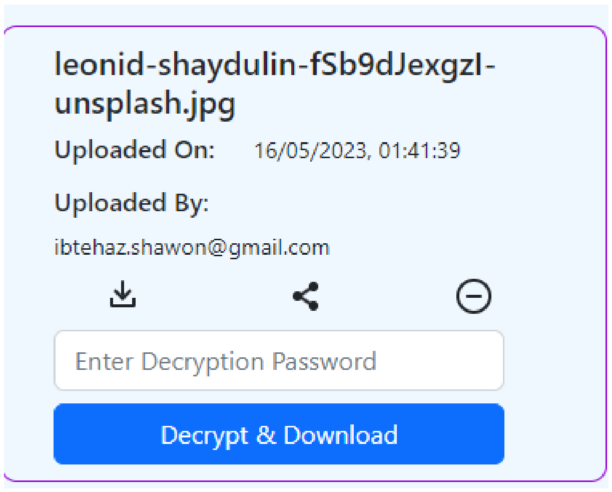
File Name: leonid-shaydulin-fSb9dJexgzI-unsplash.jpg

File Size: 1.74 MB

File Type: image/jpeg

Figure 3: After Selecting a file to upload.

On the file list, we will then see the newly uploaded file. We will be able to download the said file if the decryption is successful.


leonid-shaydulin-fSb9dJexgzI-unsplash.jpg

Uploaded On: 16/05/2023, 01:41:39

Uploaded By: ibtehaz.shawon@gmail.com

Figure 4: File Download Option

Structure of the Program

This is a full stack application, where the task is divided between frontend and backend.

Backend and Database

Language	Python
Framework	Django
Database	SQLite
Crypto Package	Cryptography
Session Management	Django Session Management
API Management	Django Rest Framework

Front End

Language	JavaScript
Framework	ReactJS
General API	Fetch
File Upload API	Axios

In the backend section, I have used SQLite as my database, to reduce the overall load on my personal computer. In future versions, this will be turned into PostgreSQL database. Since I am using Django in this project, Django maintains a very structured migration file and I have written all the query using Django's ORM. So, it's very easy to move between databases.

API End Points

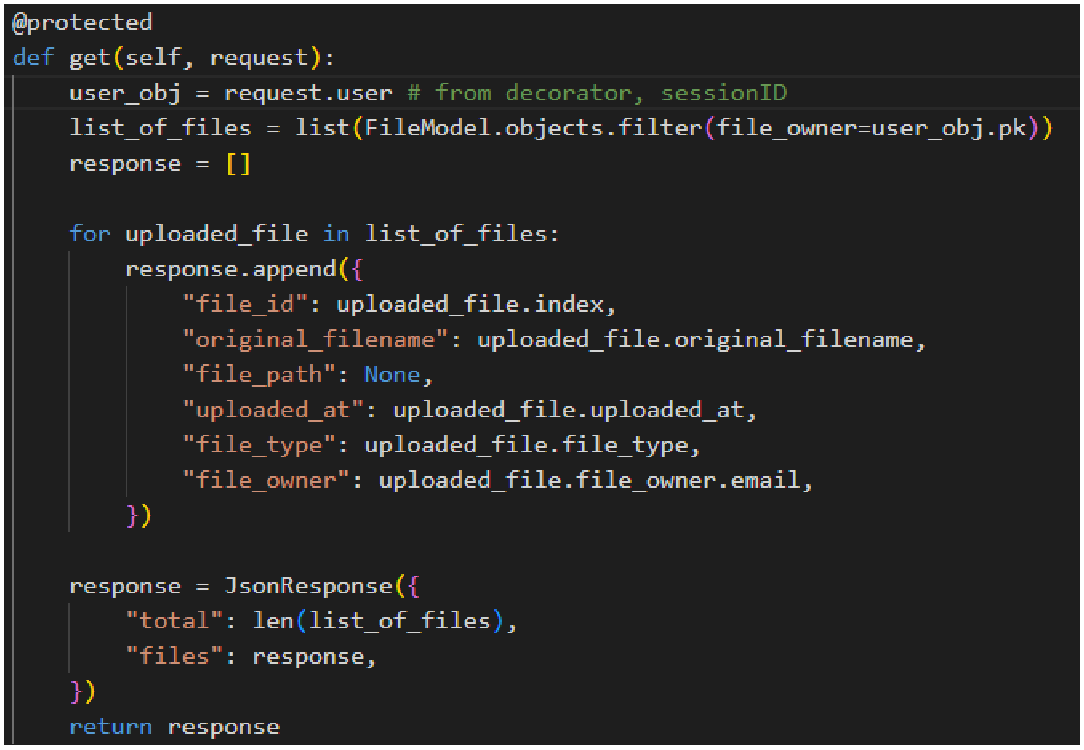
End Points	HTTP Method	Purpose	Payload	Session Included
/user/register/	POST	General Register API.	Email and password	None
/user/login/	POST	General Login API	Email and password	None
/user/logout/	POST	General Logout API	None	Yes, session ID
/share/	POST	Share an existing file with another registered user	Share_email and file_id	Yes, session ID
/share/upload/	POST	File Upload API	Mandatory parameter: {file, filename, password}. Optional Parameter:	Yes, session ID

			{shared_email, file_type}	
/share/uploaded_files/	GET	Returns a list of files uploaded by the current user	None	Yes, session ID
/share/shared_files/	GET	Returns a list of file shared with the current user	None	Yes, session ID
/share/delete_file/	DELETE	Deletes a file based on the file_id; only the owner is allowed to perform this action	File_id	Yes, session ID
/share/download/v2/<str:file_id>	GET	Deprecated API to download a file.	None	Yes, session ID
/share/download/v3/<str:file_id>	POST	Current version of API to download a decrypted file from the server.	Password	Yes, session ID

Secure Programming Solution

In this project, I have tried to complete some of the OWASP Top 10 (https://owasp.org/www-project-top-ten/) web application security risks. Here's a list of those –

1. **Broken Access Control**: In this application, users aren't allowed to perform any action without authorization. For example, users only view the file they are shared with them or own them. As you see from figure 5, only the current user's (based on session ID) file list will be loaded. Figure 6 shows the code snippet for loading shared file list, we can see, I am only loading the files shared with current user.
   Another security measure is, I have blocked unnecessary HTTP Method by default. Django by default, only accepts Get, Delete, Post, Put, and Options as request method. On API Level, each API supports only a single HTTP method and thus will automatically reject every other request. Figure 7 shows a screenshot from Insomnia API manager software. I have sent a request to /share/ API end points, but I have changed the HTTP Method to Patch. Even though, in my code I haven't handled it, Django automatically returns an error message with 405 – Method not allowed status code.
   If the session id has expired, backend will not process the requests and will return an authorization failure message with 401 status code instead.

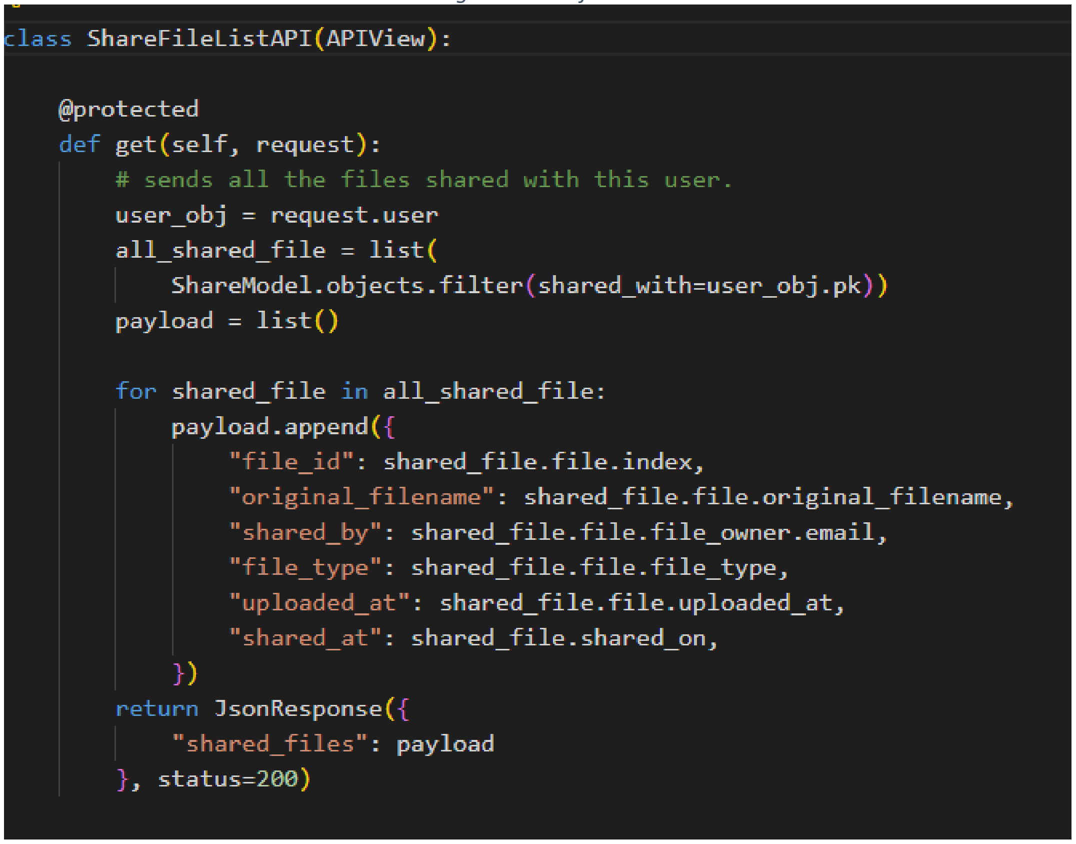


```python
@protected
def get(self, request):
    user_obj = request.user # from decorator, sessionID
    list_of_files = list(FileModel.objects.filter(file_owner=user_obj.pk))
    response = []

    for uploaded_file in list_of_files:
        response.append({
            "file_id": uploaded_file.index,
            "original_filename": uploaded_file.original_filename,
            "file_path": None,
            "uploaded_at": uploaded_file.uploaded_at,
            "file_type": uploaded_file.file_type,
            "file_owner": uploaded_file.file_owner.email,
        })

    response = JsonResponse({
        "total": len(list_of_files),
        "files": response,
    })
    return response
```

Figure 5: List of Files

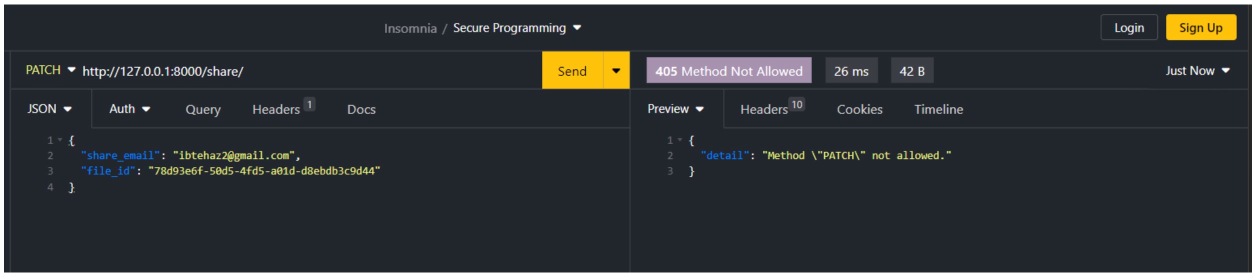


```python
class ShareFileListAPI(APIView):

    @protected
    def get(self, request):
        # sends all the files shared with this user.
        user_obj = request.user
        all_shared_file = list(
            ShareModel.objects.filter(shared_with=user_obj.pk))
        payload = list()

        for shared_file in all_shared_file:
            payload.append({
                "file_id": shared_file.file.index,
                "original_filename": shared_file.file.original_filename,
                "shared_by": shared_file.file.file_owner.email,
                "file_type": shared_file.file.file_type,
                "uploaded_at": shared_file.file.uploaded_at,
                "shared_at": shared_file.shared_on,
            })
        return JsonResponse({
            "shared_files": payload
        }, status=200)
```

Figure 6: Shared List API



```
PATCH http://127.0.0.1:8000/share/
```

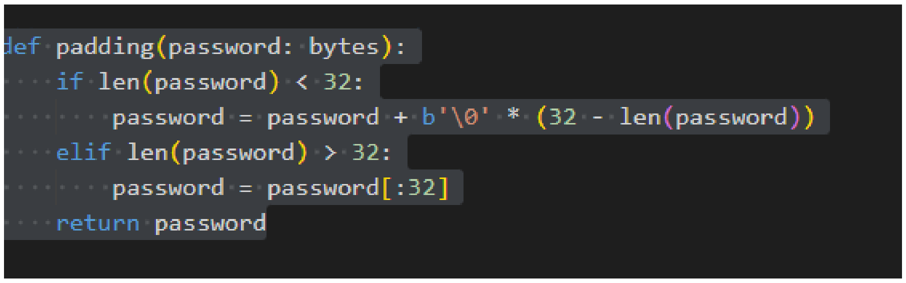
```
{
  "share_email": "ibtehaz2@gmail.com",
  "file_id": "78d93e6f-50d5-4fd5-a01d-d8ebdb3c9d44"
}
```

```
405 Method Not Allowed   26 ms   42 B
```

```
{
  "detail": "Method \"PATCH\" not allowed."
}
```

Figure 7: Automatic 405 - Method not allowed response

2. **Cryptographic Failures**: In Django, all passwords are hashed with by default PBKDF2 SHA256. All files are encrypted with AES-CBC-128 and passwords are selected from 8 – 32 characters, there has to be at least one number, one upper case, one lower case and one special character. Special characters are chosen from this list !@#$%^&*(),.?:{}|<>. Python's cryptography package requires 32 characters as password for CBC and its fixed if I use Fernet. Hence, there's a padding required, remaining bytes are filled with 0s. Both backend and front end has been deployed in the internet over HTTPS as test basis. They are available at https://1234.ibtehaz.xyz and https://tuni-projects.github.io/Encrypted-Signal/. Both of these sites enforce HTTPS, however, at the time of this report writing they are fully operational, they will be operational by May 17.

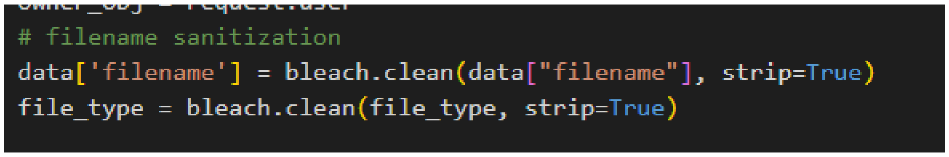


```
def padding(password: bytes):
    if len(password) < 32:
        password = password + b'\0' * (32 - len(password))
    elif len(password) > 32:
        password = password[:32]
    return password
```

Figure 8: Password Padding

3. **Injection**: To protect against injection, all user supplied data are sanitized. I have used a python package, Bleach (https://pypi.org/project/bleach/) to sanitize the incoming data. Bleach only does HTML sanitization (see Figure 9)
   In the case of SQL Injection, I let Django handles that part as Django does it well. This is a snippet from Django's documentation (version 4.2)

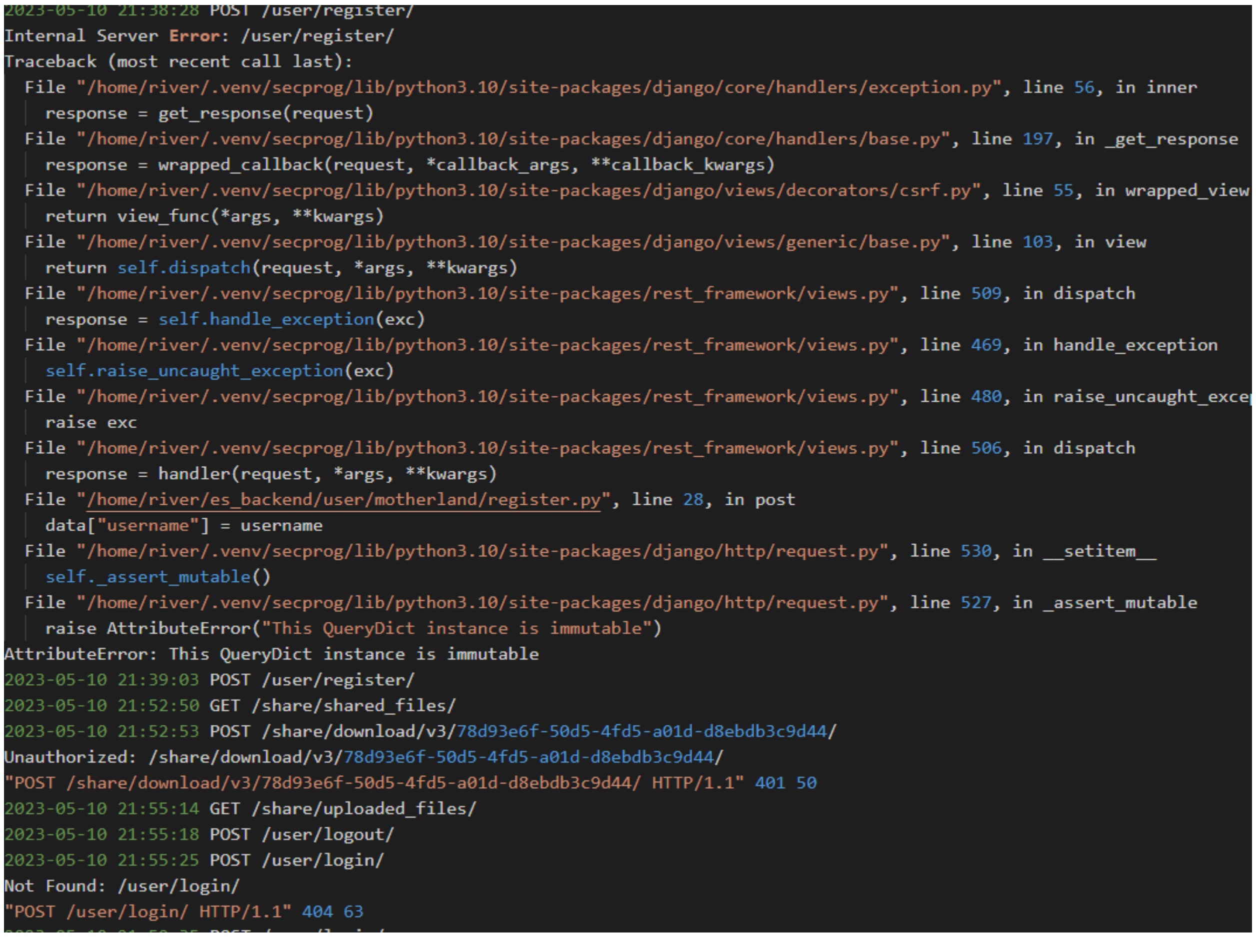
   Django's querysets are protected from SQL injection since their queries are constructed using query parameterization. A query's SQL code is defined separately from the query's parameters. Since parameters may be user-provided and therefore unsafe, they are escaped by the underlying database driver.



```
owner_obj = request.user
# filename sanitization
data['filename'] = bleach.clean(data["filename"], strip=True)
file_type = bleach.clean(file_type, strip=True)
```

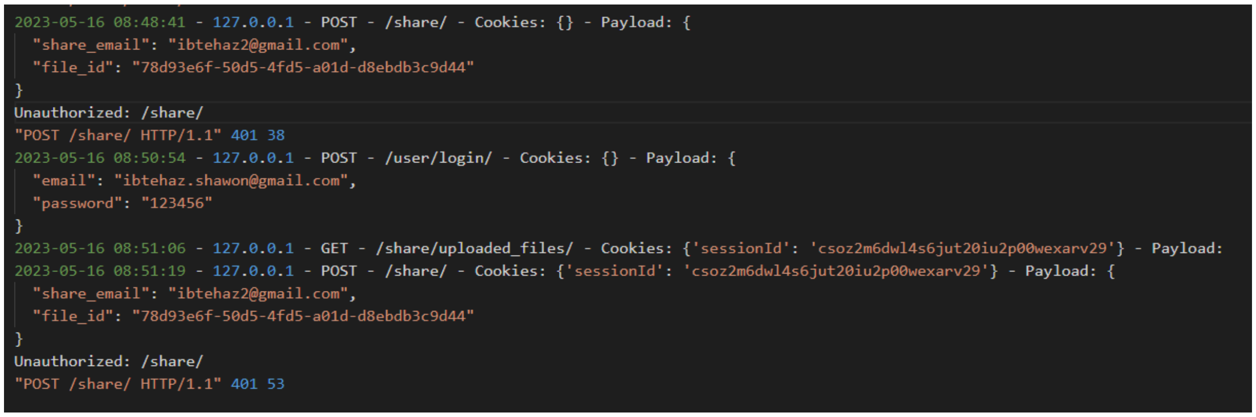
Figure 9: HTML Sanitization by Bleach

4. Security Logging and Monitoring Failures: In the Backend, all incoming requests (accepted or not) generate a log. The older version of logs only kept the details of when the request happened, in which endpoint. Also, there are logs to keep track of when the application crashed. Figure 10 is a snippet of the log file. Currently, the log format is current date – IP address – HTTP method – endpoint – Cookies: {} – Payload: {} – details of request. Figure 11 has the snippet of that log file.



```
2023-05-10 21:38:28 POST /user/register/
Internal Server Error: /user/register/
Traceback (most recent call last):
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/django/core/handlers/exception.py", line 56, in inner
    response = get_response(request)
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/django/core/handlers/base.py", line 197, in _get_response
    response = wrapped_callback(request, *callback_args, **callback_kwargs)
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/django/views/decorators/csrf.py", line 55, in wrapped_view
    return view_func(*args, **kwargs)
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/django/views/generic/base.py", line 103, in view
    return self.dispatch(request, *args, **kwargs)
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/rest_framework/views.py", line 509, in dispatch
    response = self.handle_exception(exc)
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/rest_framework/views.py", line 469, in handle_exception
    self.raise_uncaught_exception(exc)
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/rest_framework/views.py", line 480, in raise_uncaught_exce
    raise exc
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/rest_framework/views.py", line 506, in dispatch
    response = handler(request, *args, **kwargs)
  File "/home/river/es_backend/user/motherland/register.py", line 28, in post
    data["username"] = username
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/django/http/request.py", line 530, in __setitem__
    self._assert_mutable()
  File "/home/river/.venv/secprog/lib/python3.10/site-packages/django/http/request.py", line 527, in _assert_mutable
    raise AttributeError("This QueryDict instance is immutable")
AttributeError: This QueryDict instance is immutable
2023-05-10 21:39:03 POST /user/register/
2023-05-10 21:52:50 GET /share/shared_files/
2023-05-10 21:52:53 POST /share/download/v3/78d93e6f-50d5-4fd5-a01d-d8ebdb3c9d44/
Unauthorized: /share/download/v3/78d93e6f-50d5-4fd5-a01d-d8ebdb3c9d44/
"POST /share/download/v3/78d93e6f-50d5-4fd5-a01d-d8ebdb3c9d44/ HTTP/1.1" 401 50
2023-05-10 21:55:14 GET /share/uploaded_files/
2023-05-10 21:55:18 POST /user/logout/
2023-05-10 21:55:25 POST /user/login/
Not Found: /user/login/
"POST /user/login/ HTTP/1.1" 404 63
```

Figure 10: Log file



```
2023-05-16 08:48:41 - 127.0.0.1 - POST - /share/ - Cookies: {} - Payload: {
  "share_email": "ibtehaz2@gmail.com",
  "file_id": "78d93e6f-50d5-4fd5-a01d-d8ebdb3c9d44"
}
Unauthorized: /share/
"POST /share/ HTTP/1.1" 401 38
2023-05-16 08:50:54 - 127.0.0.1 - POST - /user/login/ - Cookies: {} - Payload: {
  "email": "ibtehaz.shawon@gmail.com",
  "password": "123456"
}
2023-05-16 08:51:06 - 127.0.0.1 - GET - /share/uploaded_files/ - Cookies: {'sessionId': 'csoz2m6dwl4s6jut20iu2p00wexarv29'} - Payload:
2023-05-16 08:51:19 - 127.0.0.1 - POST - /share/ - Cookies: {'sessionId': 'csoz2m6dwl4s6jut20iu2p00wexarv29'} - Payload: {
  "share_email": "ibtehaz2@gmail.com",
  "file_id": "78d93e6f-50d5-4fd5-a01d-d8ebdb3c9d44"
}
Unauthorized: /share/
"POST /share/ HTTP/1.1" 401 53
```

Figure 11: Updated Log File

5. **Identification and Authentication Failures**: In my application, all the session related information is passed through headers. Now, the session ID is not HTTP Only, even though I am not accessing the session ID using JavaScript. However, I don't want to change the session ID to HTTP Only without prior testing, as it might break the system.
   In the application, there are currently two ways a session id can become invalid.
   a. If the session ID expired normally.
   b. If a user logged out

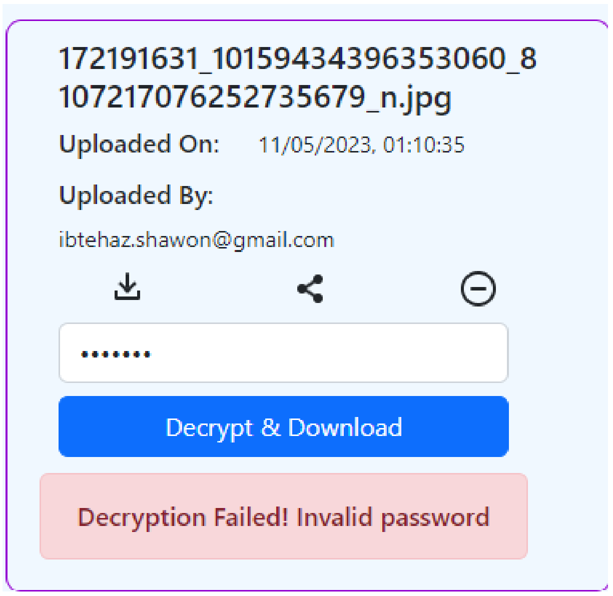
   Currently, there's no process to determine if there's any attack originated either from this user's credential or on the overall application, so there's no way to invalidate a single session ID or a batch. In future versions, this will certainly happen.
   All passwords must be 8 – 32 characters long and a minimum of one uppercase, one lowercase and one special character and one number are required. It beats the common password attack vector. However, there's still a common password validator in code to check against Django's built-in common password list. I have used **django.contrib.auth.password_validation** class's **validate_password** function to validate the password against all password validator middleware.

6. **Security Misconfiguration**: The application does not show any stack trace on the front-end. The front-end receives an appropriate error message for every kind of error, even if it's unknown. Figure 12 shows an example of such a message. There are some validations on the front-end, however most of them are vanity checks. Backend has more robust validation checks against every know issues. In both front-end and back-end, I have used the best security practices. Moreover, every request from server comes with these headers -

   Date: Tue, 16 May 2023 09:43:02 GMT
   Server: WSGIServer/0.2 CPython/3.10.6
   Content-Type: application/json
   SameSite: None
   Vary: Accept, Cookie, Origin
   Allow: POST, OPTIONS
   X-Frame-Options: DENY
   Content-Length: 71
   X-Content-Type-Options: nosniff

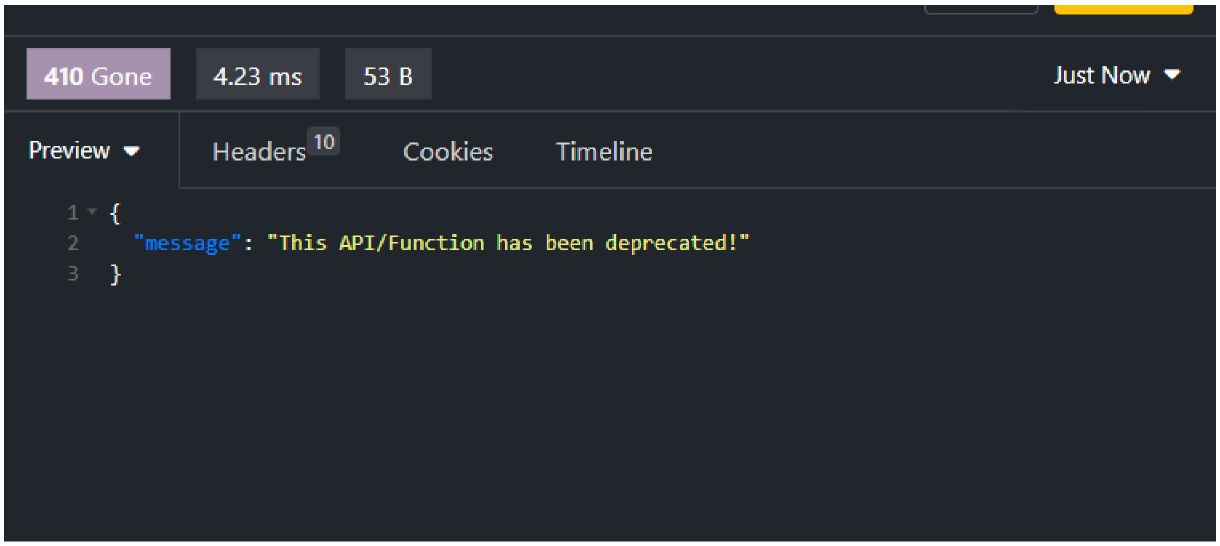
Referrer-Policy: same-origin
Cross-Origin-Opener-Policy: same-origin
Set-Cookie: sessionId=apofw8t22mty2h8kkpyyafruulzcantn; expires=Tue, 16 May 2023 21:43:02 GMT; Max-Age=43200; Path=/; SameSite=None; Secure

These headers are sent from localhost, on nginx server, some of the parameters might be different. Finally, there's no default accounts enabled on both front-end and back-end. So, there's no super admin enabled, either during the development phase or production phase. All the API end points are active now, except for File Download API v2. However, if anyone tries to use that, it will throw a deprecation error message.


172191631_10159434396353060_8107217076252735679_n.jpg
Uploaded On: 11/05/2023, 01:10:35
Uploaded By:
ibtehaz.shawon@gmail.com

Decryption Failed! Invalid password

Figure 12: Decryption Failed Error Message



```
{
  "message": "This API/Function has been deprecated!"
}
```

Figure 13: Depecration Error Message for Download API v2

7. **Server-Side Request Forgery (SSRF)**: Basically, I don't fetch remote resources from user-supplied URL.

Testing

Manual Testing

Manual testing was done both on front-end and back-end using Insomnia API Manager Software. Testing was done using various parameters to see if the desired output is produced. A lot of bugs have been mitigated through these testing. All API end points have been tested thoroughly over the course of development. The first issue I found was not getting the proper session id, after login. It was working on postman; however, it wasn't working on the browser. After careful inspection, I found the issue on how I was setting the cookie from the backend and fixed that issue. Another set of issues I have found was front-end crashing for invalid input or if the backend crashes. So, in the backend, all the known types of issues will return a JSON message and front-end will show that message. In the fetch/axios block, all the codes have an error handler present as well.

Automatic Testing

I have written some automatic Unit testing for backend; however, they are not working how they were supposed to be. So, there's nothing to report in this case.

Known Issues/Vulnerabilities

One of the known issues that I didn't get the time to resolve is "what will happen if session id expired in the middle of a user session". For example, if the session id expired when the user was not on the website, then the next time user comes to use the application, it will automatically show the login page. However, in the middle of the session, it will show authorization error message every time there's an error with session id but I would have liked to initiate an force logout in this case. Another security issue, I think exists because the session id is active for too long, one session id is used in multiple requests. I would have liked to issue a new session id with every request (success/failed). This doesn't require any serious coding; however, I didn't get the time to finish this work.

Future Development

1. Implementing Encryption & Decryption at the front-end
2. Notification Management
3. Password Recovery Option
4. User Profile Management
5. Auto Suggestions for Share by Email field
6. Unit Testing at both Backend and Front End
7. Session Expiry Manager at Frontend