

Project: Wagtail

([Repo Link](#))

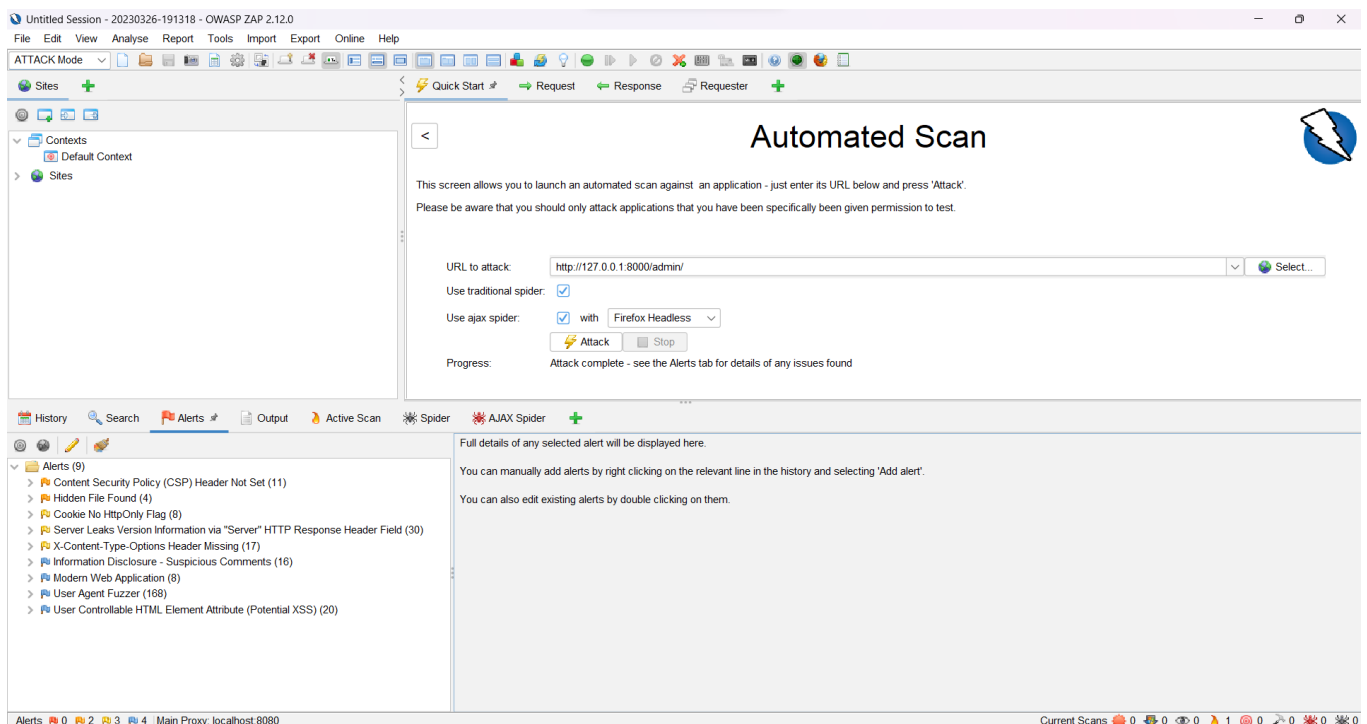
Phase 1 Report

Part 1: OWASP Top Ten / Ethical Hack

We first ran a ZAP Automated scan on the endpoint 127.0.0.1:8000/admin

It returned some alerts regarding some vulnerabilities it found during the scan.

Here's a screenshot –



None of the alerts were high risk, however some of the interesting ones were –

Cookie No HttpOnly Flag – There were a few cookies being sent without using the HttpOnly flag. This could lead to an attacker accessing the cookie using JavaScript, which may lead to session hijacking.

There was also information leakage which could lead to the attacker getting to know the server versions of the services running and exploit the vulnerabilities on those specific versions.

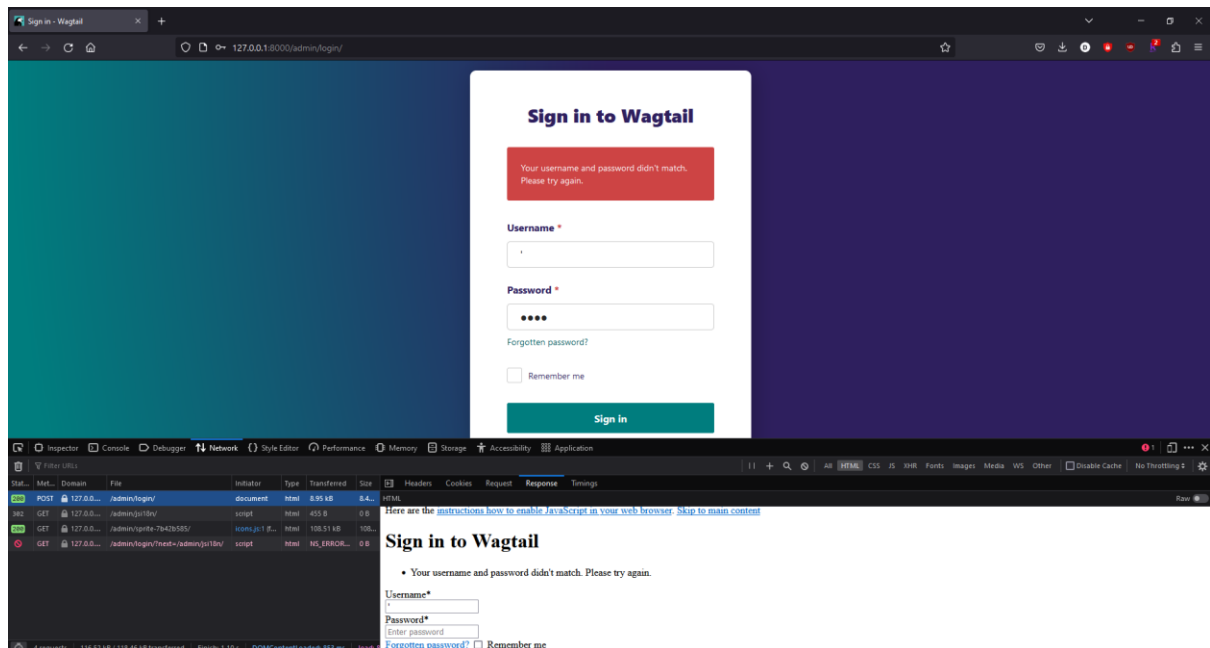
It also alerted for suspicious comments but after analyzing those comments there was no information leakage and they didn't provide any info which the attacker could exploit.

1) SQL Injection

Firstly, we tried to use ZAP to perform fuzzing, however Django gave CORS error when proxying via ZAP. So, we decided to perform the attacks manually.

We tried performing SQL injection on the login form.

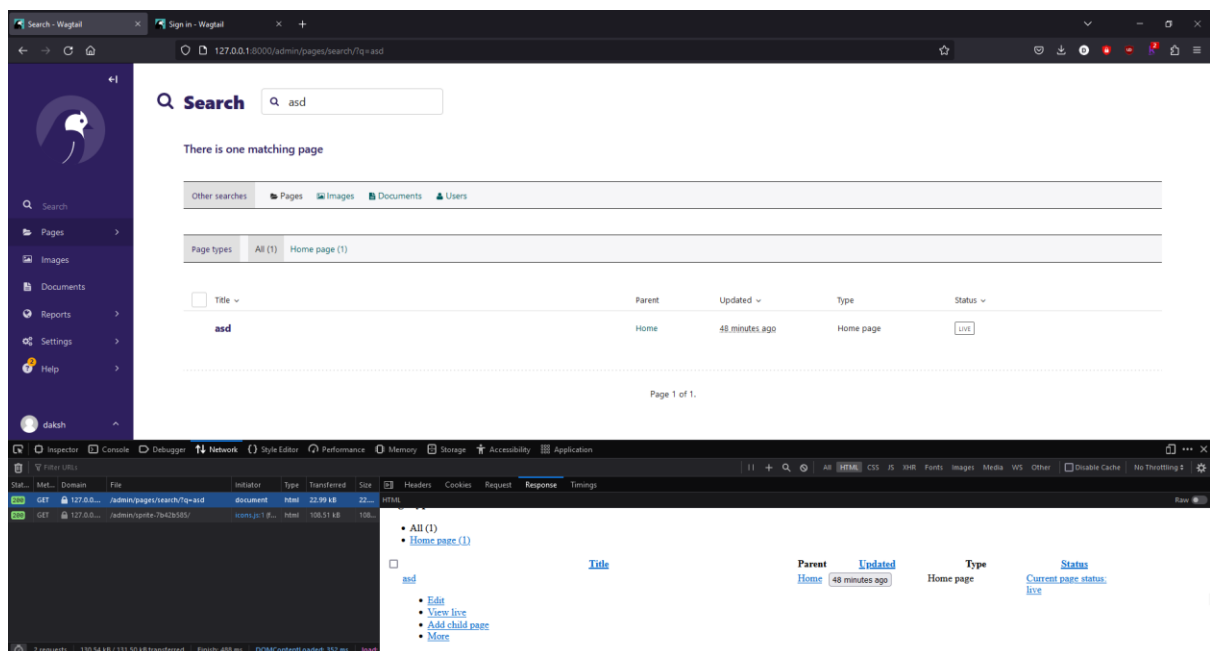
First, we tried entering a simple apostrophe and checked the response to our request in the developer tools to see if any information is returned that can be exploited.



No information was obtained here.

We then tried to perform a sql tautology by entering a valid username and " pass' or 2=2 " as the password to see if this would result in anything but this didn't work either.

We further tried exploring other possible endpoints which can be attacked for SQL injection but none of them worked.

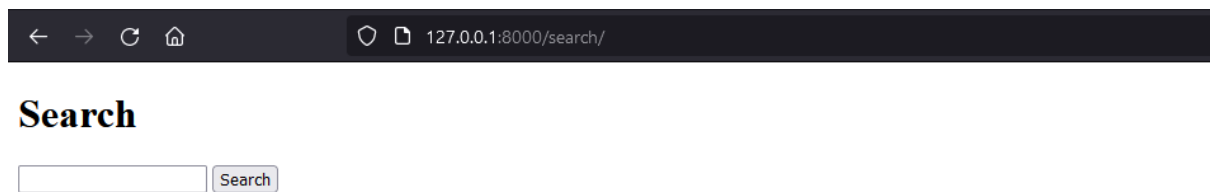


The developers of wagtail have used the Django framework to build their application. Django is known to be resilient against majority of SQL attacks. This is because the queries are generated using query parameterization. Thus, the developers are using a well known framework which is robust against majority of SQL injection attacks.

2) Security Misconfiguration

We first tried to access the Django superuser panel by going to the URL - <http://127.0.0.1:8000/admin> which is the superuser panel by default for Django. We tried various well known administrator user id and password combos (admin,admin ; admin,password ; etc) none of them worked. We then played around with the URL trying different endpoints.

When visiting the URL <http://127.0.0.1:8000/search> we were returned the following -



We entered a script tag in the search bar to see what it would result in.

OperationalError at /search/

```
fts5: syntax error near "<" The original query was: SELECT "wagtailsearch_indexentry_fts"."autocomplete", "wagtailsearch_indexentry_fts"."title", "wagtailsearch_indexentry_fts"."body", "wagtailsearch_indexentry_fts"."rowid", "wagtailsearch_indexentry"."id", "wagtailsearch_indexentry"."content_type_id", "wagtailsearch_indexentry"."object_id", "wagtailsearch_indexentry"."title_norm", "wagtailsearch_indexentry"."autocomplete", "wagtailsearch_indexentry"."title", "wagtailsearch_indexentry"."body" FROM "wagtailsearch_indexentry_fts" INNER JOIN "wagtailsearch_indexentry" ON ("wagtailsearch_indexentry_fts"."rowid" = "wagtailsearch_indexentry"."id") WHERE (wagtailsearch_indexentry_fts MATCH %s AND "wagtailsearch_indexentry"."content_type_id" IN (%s, %s)) ORDER BY bm25(wagtailsearch_indexentry_fts) DESC("title body") : ("attacked!")</script> AND "<script>alert('You AND "have AND "been')", 1, 2)

Request Method: GET
Request URL: http://127.0.0.1:8000/search/?query=%3Cscript%3Ealert%28%22You+have+been+attacked%21%22%29%3C%2Fscript%3E
Django Version: 4.1.7
Exception Type: OperationalError
Exception Value: fts5: syntax error near "<" The original query was: SELECT "wagtailsearch_indexentry_fts"."autocomplete", "wagtailsearch_indexentry_fts"."title", "wagtailsearch_indexentry_fts"."body", "wagtailsearch_indexentry_fts"."rowid", "wagtailsearch_indexentry"."id", "wagtailsearch_indexentry"."content_type_id", "wagtailsearch_indexentry"."object_id", "wagtailsearch_indexentry"."title_norm", "wagtailsearch_indexentry"."autocomplete", "wagtailsearch_indexentry"."title", "wagtailsearch_indexentry"."body" FROM "wagtailsearch_indexentry_fts" INNER JOIN "wagtailsearch_indexentry" ON ("wagtailsearch_indexentry_fts"."rowid" = "wagtailsearch_indexentry"."id") WHERE (wagtailsearch_indexentry_fts MATCH %s AND "wagtailsearch_indexentry"."content_type_id" IN (%s, %s)) ORDER BY bm25(wagtailsearch_indexentry_fts) DESC("title body") : ("attacked!")</script> AND "<script>alert('You AND "have AND "been')", 1, 2)
Exception Location: C:\wagtail\env\lib\site-packages\django\search\backends\database\sqlite\sqlite.py, line 548, in search
Raised during: search.views.search
Python Executable: C:\wagtail\env\Scripts\python.exe
Python Version: 3.10.7
Python Path: ['C:\\wagtail\\env\\site', 'C:\\Python\\python310.zip', 'C:\\Python\\DLLs', 'C:\\Python\\lib', 'C:\\Python', 'C:\\wagtail\\env', 'C:\\wagtail\\env\\lib\\site-packages']
Server time: Tue, 28 Mar 2023 21:54:15 +0000
```

Traceback [switch to copy-and-paste view](#)

```
C:\wagtail\env\lib\site-packages\django\db\backends\utils.py, line 89, in _execute
89.         return self.cursor.execute(sql, params)

> Local vars

C:\wagtail\env\lib\site-packages\django\db\backends\sqlite\base.py, line 357, in execute
357.         return Database.Cursor.execute(self, query, params)

> Local vars
```

The above exception (fts5: syntax error near "<") was the direct cause of the following exception:

```
C:\wagtail\env\lib\site-packages\wagtailsearch\backends\database\sqlite\sqlite.py, line 544, in search
544.         obj_ids = [

> Local vars

C:\wagtail\env\lib\site-packages\django\models\query.py, line 394, in _fetch__
394.         self._fetch_all()

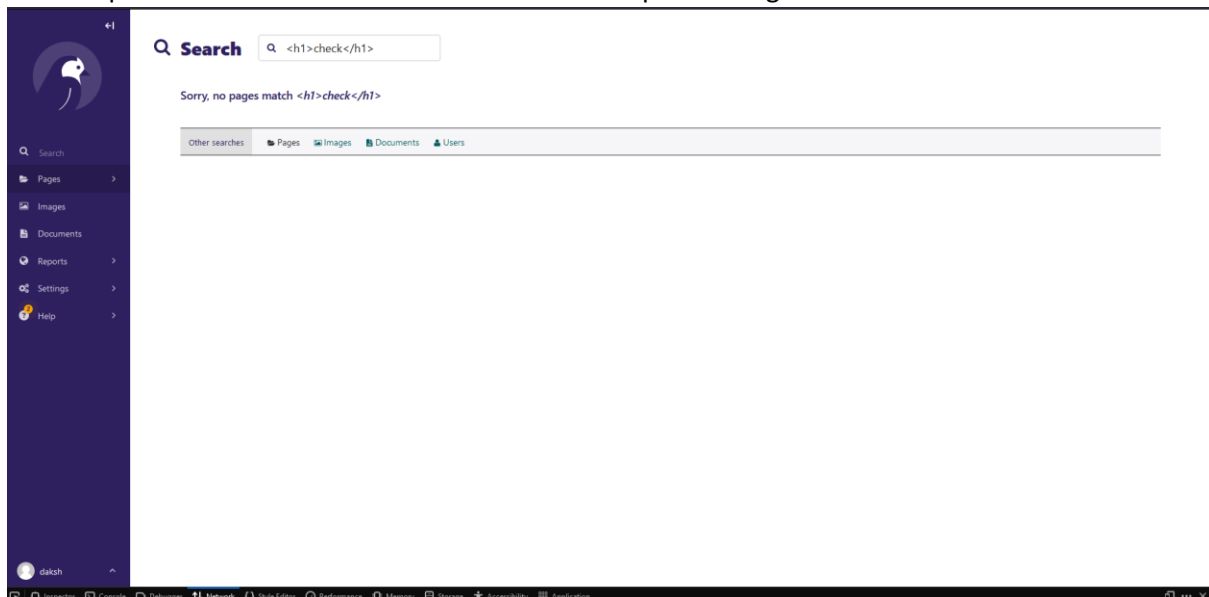
> Local vars
```

This is what was returned. Here we can see a lot of sensitive information being exposed. The SQL query structure can be seen and also the version of Python and Django being run is visible too. On scrolling down, the entire stack trace can be seen too. An attacker can exploit all this sensitive information being exposed to craft a serious attack on the system.

This can be avoided by properly configuring the options present in Django framework. An error message or a default page can be displayed when a user is trying to access something which results in an error on the backend.

3) XSS – Cross Side Scripting

We first performed a basic XSS attack to see if user input is being validated or not.

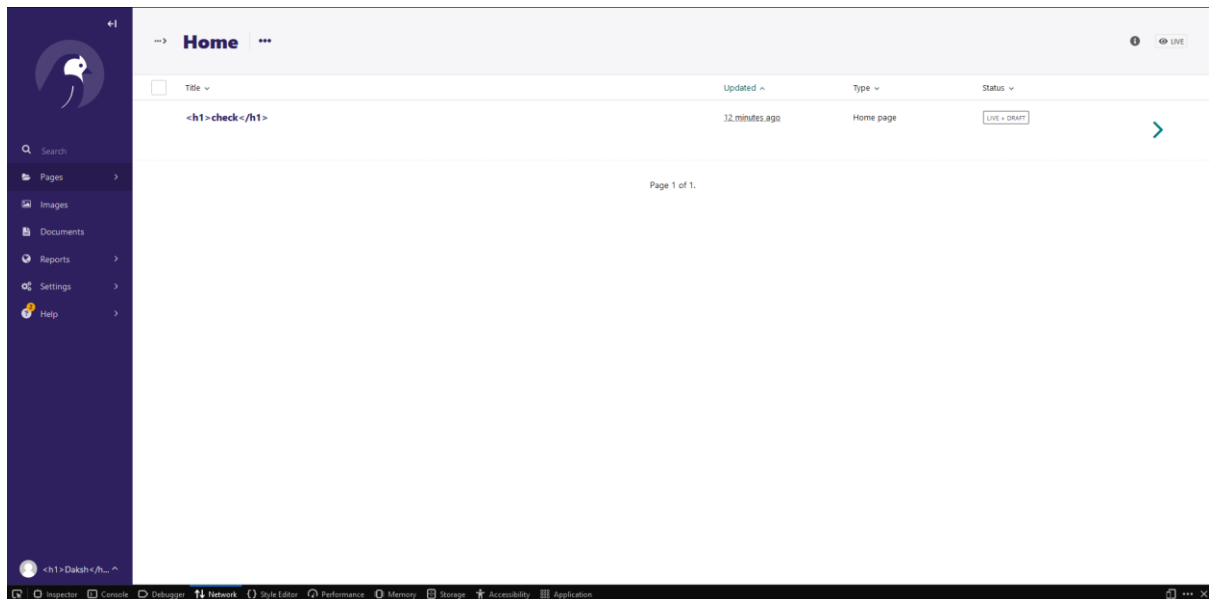


This showed that the application is performing input sanitization and is not prone to reflected XSS attack.

So we try to perform a persisted XSS attack to see if it is vulnerable to that.

We inputted the page name with a basic <h1> tag but this didn't work either.

We tried at various input places to perform an XSS attack but it seems like the developers have taken XSS into consideration while developing wagtail.



A few ways the developers have protected against this type of attack could be that they are performing proper input sanitation checks and using escape characters to replace characters such as '<'/>' so that they are not confused for commands while generating the webpage. They have also implemented these checks both on the server as well as client side.

4 – Identification and Authentication

We first created a page using wagtail and then viewed our request in the developer tools to see if the post request body mentioned the author so that we could try to change the author name by sending a request via the console. However, no such property was there in the request body.

We then tried to send login requests frequently to see if wagtail times out after a series of unsuccessful login requests to block out attackers.

We used the following script -

```
>> for (let i = 1; i <= 100; i++){
  fetch("http://127.0.0.1:8000/admin/login/", {
    "credentials": "include",
    "headers": {
      "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/111.0",
      "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8",
      "Accept-Language": "en-US,en;q=0.5",
      "Content-Type": "application/x-www-form-urlencoded",
      "Upgrade-Insecure-Requests": "1",
      "Sec-Fetch-Dest": "document",
      "Sec-Fetch-Mode": "navigate",
      "Sec-Fetch-Site": "same-origin",
      "Sec-Fetch-User": "?1"
    },
    "referrer": "http://127.0.0.1:8000/admin/login/",
    "body": "csrfmiddlewaretoken=OHTfa0s42j3lxmum6Em8FZq3BunaaVBwr1KAext6jDL7I1UPKani18nQeM8ur6R7x&next=%2Fadmin%2F&username=123&password=123",
    "method": "POST",
    "mode": "cors"
  }).then(response => {
    if (response.ok) {
      console.log(`Attempt ${i}: Login successful`);
    } else {
      console.log(`Attempt ${i}: Login failed`);
    }
  })
  .catch(error => {
    console.error(error);
  });
}
```

The script ran successfully. It sent an invalid login request 100 times. The expected behaviour was that our requests would get timed out after a certain amount of unsuccessful login requests. However, the response to our requests was 200 OK.

| Status | Method | Domain | File | Initiator | Type | Transferred | Size | Request | Response | Timings | Stack Tra |
|--------|--------|----------------|---------------|------------|------|-------------|---------|--|----------|---------|-----------|
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | Filter Request Parameters | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | Form data csrfmiddlewaretoken: "OHTfa0s42j3lxmum6Em8FZq3BunaaVBwr1KAext6jDL7I1UPKani18nQeM8ur6R7x&next=%2Fadmin%2F&username=123&password=123" | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |
| 200 | POST | 127.0.0.1:8000 | /admin/login/ | /2 (fetch) | html | 8.95 kB | 8.41 kB | | | | |

An attacker can use a brute-force algorithm to try and gain access to the admin control panel due to this.

This potential attack can be avoided by timing out requests from a particular IP after a certain number of unsuccessful login attempts.

A Captcha can also be used so that each time a new request is sent it requires a new captcha so that the process of sending requests cannot be automated.

5 – Cryptographic Failures

Cryptographic failures refer to vulnerabilities or weaknesses in cryptographic systems, algorithms, or protocols that can lead to the compromise of sensitive information or the ability for an attacker to impersonate a legitimate user. This is marked as one of the most common vulnerabilities in the latest OWASP Top 10 list.

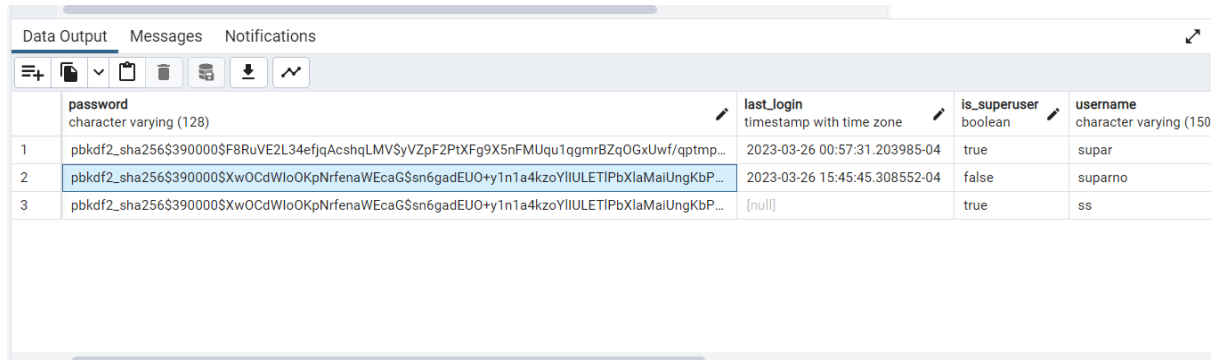
To check if cryptographic Failure vulnerability is present in the project, we studied the password which is being stored in the Database for each user. We found the user password in the following format-

`pbkdf2_sha256$390000$XwOCdWloOKpNrfenaWEcaG$sn6gadEUO+y1n1a4kzoYIIULETIPbXlaMaiUngKbPr8=`

On studying the encryption algorithm from the Prefix added in the hashed output, we found that the system is using PBKDF2 algorithm with SHA256 as the underlying hash function.

As this is a one-way process can cannot be reverted, so it is impossible to decrypt the password.

Also we tried copying the hash value from the password column of one user into another just to find that the same password is still not working- which suggests that along with the password entered by the user, the system is adding some other factors or user details before hashing the password and it is matching the same while trying to validate the password while login.



| | password character varying (128) | last_login timestamp with time zone | is_superuser boolean | username character varying (150) |
|---|---|--|-------------------------|-------------------------------------|
| 1 | pbkdf2_sha256\$390000\$F8RuVE2L34efjqAcshqLMV\$yVZpF2P+XFg9X5nFMUqu1qgmrBZqOGxUwf/qptmp... | 2023-03-26 00:57:31.203985-04 | true | supar |
| 2 | pbkdf2_sha256\$390000\$XwOCdWloOKpNrfenaWEcaG\$sn6gadEUO+y1n1a4kzoYIIULETIPbXlaMaiUngKbP... | 2023-03-26 15:45:45.308552-04 | false | suparno |
| 3 | pbkdf2_sha256\$390000\$XwOCdWloOKpNrfenaWEcaG\$sn6gadEUO+y1n1a4kzoYIIULETIPbXlaMaiUngKbP... | [null] | true | ss |

Using Zap also we ran an interception test on the application and tried logging in into the system and perform some of the tasks such as uploading files and creating new pages. We couldn't find any alert in ZAP regarding Cryptographic Failures.

Part 2: Static Analysis

Since our code is in Python, we have used the Bandit tool to scan for vulnerabilities in our codebase. Furthermore, we have used the OWASP Dependency Checker to scan the code of vulnerabilities.

Changes for the System Developers:

1. Moment.js

Description: It is a JavaScript date library for parsing, validating, manipulating, and formatting dates.

Mitigation: It contains a path traversal vulnerability, which is patched in version 2.29.2 and later. The current version being used in the project is 2.20.1. Other way to tackle this problem is to sanitize the user provided locale name before passing it to Moment.js

Effect: On change, a path traversal attack can be mitigated.

Issue Report: <https://github.com/moment/moment/security/advisories/GHSA-wc69-rhjr-hc9g>

2. SnakeYaml

Description: SnakeYAML is a YAML-parsing library with a high-level API for serialization and deserialization of YAML documents

Mitigation: SnakeYaml's Constructor() class does not restrict the type of an object after deserialization, which lets an attacker run arbitrary code if they have control of the YAML document. To tackle this, we can make use of SafeConstructor to restrict deserialization. It prevents an attacker from accessing the classpath by default. When instantiating the `Constructor` or `SafeConstructor`, you must pass a `LoaderOptions` object where one can further set parsing restrictions. By default, all global tags are now blocked.

Effect: On change, remote code execution would be mitigated. It prevents an attacker from specifying arbitrary tags in the YAML document which then get converted into Java "gadgets".

Issues Documentation: <https://www.veracode.com/blog/research/resolving-cve-2022-1471-snekeyaml-20-release-0>

3. H2 Database Engine

Description: H2 has a web-based admin console that can be started via the CLI. One of the arguments is "-webAdminPassword", which allows the user to specify the password in plaintext for the web admin console.

Mitigation: We can mitigate this vulnerability by removing the "-webAdminPassword" as a valid CLI argument. Furthermore, we can improve the security is to read the CLI arguments and after that set the values provided by the loadProperties () function.

Effect: On fixing this issue, we can prevent an attacker that has obtained local access through some means from getting the password for the H2 web admin console by looking at the running processes. It also prevents the user on a system running H2 that has been started with the webAdminPassword argument from getting the administrator password and escalating their privileges to perform any action on the system that the owner of the H2 process would be able to.

Issue Reference: <https://github.com/h2database/h2database/issues/1294>

4. MarkSafe Function

Description: The `mark_safe()` function marks the returned content as “safe to render”. This instructs the template engine to bypass HTML escaping, creating the possibility of a XSS vulnerability.

Mitigation: To tackle this, we avoid the usage of this function. Instead, we make the use of `SafeString` method. Functions and classes for working with “safe strings”: strings that can be displayed safely without further escaping in HTML. Marking something as a “safe string” means that the producer of the string has already turned characters that should not be interpreted by the HTML engine (e.g. ‘<’) into the appropriate entities.

Effect: With this, we can prevent cross site scripting attack. We prevent the embedded script being passed as safe to render and carry on with the rest of the processes.

Issue Reference: https://bandit.readthedocs.io/en/1.7.5/plugins/b703_django_mark_safe.html

5. Guava – Creation of Temporary File in Directory with Insecure Permissions

Description: Guava is a suite of core and expanded libraries. This software creates temporary files which have insecure permissions. Users having system access can access these resources easily.

Mitigation: These files are stored in the `/temp` directory which is accessible by all users. This is the default directory and can be set to an actual safe directory. We can do this by explicitly setting the “`java.io.tmpdir`” system property to that safe directory when starting the JVM.

Effect: With this change, we can prevent the attackers having access to the system from accessing these records from the `/temp` directory

Issues Reference: <https://github.com/google/guava/issues/4011>

Part 3: Security Requirements

Develop a list of ten security requirements to the system. Map these requirements back to one or more of the security objects (e.g., confidentiality). Ensure the requirements are "SMART" (e.g., specific)

1. Requirement: All user passwords must be hashed and salted. SHA-256 or above should be used to hash passwords.

Objective: Authentication and Access Control - Prevent unauthorized access to user accounts and sensitive information.

2. Requirement: HTTPS/TLS encryption must be enabled for all web traffic. Allowlist-Denylist should be implemented to restrict unwanted traffic into the system.

Objective: Confidentiality and Integrity - Ensure that sensitive information cannot be intercepted or modified during transmission.

3. Requirement: Access to sensitive data (e.g. user information, payment information) must be encrypted at rest when transmitting through API and should be decrypted at source.

Objective: Confidentiality and Integrity - Ensure that sensitive data cannot be accessed or modified by unauthorized parties.

4. Requirement: User input must be validated and sanitized to prevent injection attacks. Validations should be added both in front-end as well as back-end. Prepared statements or Stored Procedure should also be implemented to execute DB queries using user Input.

Objective: confidentiality, integrity, and availability - Protect against injection attacks that can execute malicious code or manipulate data.

5. Requirement: Role-based access control (RBAC) must be implemented to limit access to sensitive data and features.

Objective: Authorization - Ensure that users can only access the resources and functions that they are authorized to access.

6. Requirement: Audit logs must be enabled and reviewed regularly to detect and respond to security incidents.

Objective: Accountability and Identification and Authentication This provides a clear audit trail that can be used to hold users accountable for their actions and help identify any unauthorized activity that may have occurred.

7. Requirement: Password policies must be enforced (e.g. minimum length, complexity requirements). Password should be minimum 12 character long and maximum 128 characters long.

Objective: Authentication and Access Control - Prevent unauthorized access to user accounts and sensitive information.

8. Requirement: Cross-site scripting (XSS) protection must be implemented to prevent injection of malicious scripts. Inputs should be validated, and static analysis tools can be implemented to prevent this attack.

Objective: confidentiality, integrity, and availability - Protect against injection attacks that can execute malicious code or manipulate data resulting in leaking of sensitive information, deletion of data or improper modification of Data.

9. Requirement: Two-factor authentication (2FA) must be implemented for all user accounts. Password combined with a one-time Token sent to the email address can be used as 2FA.

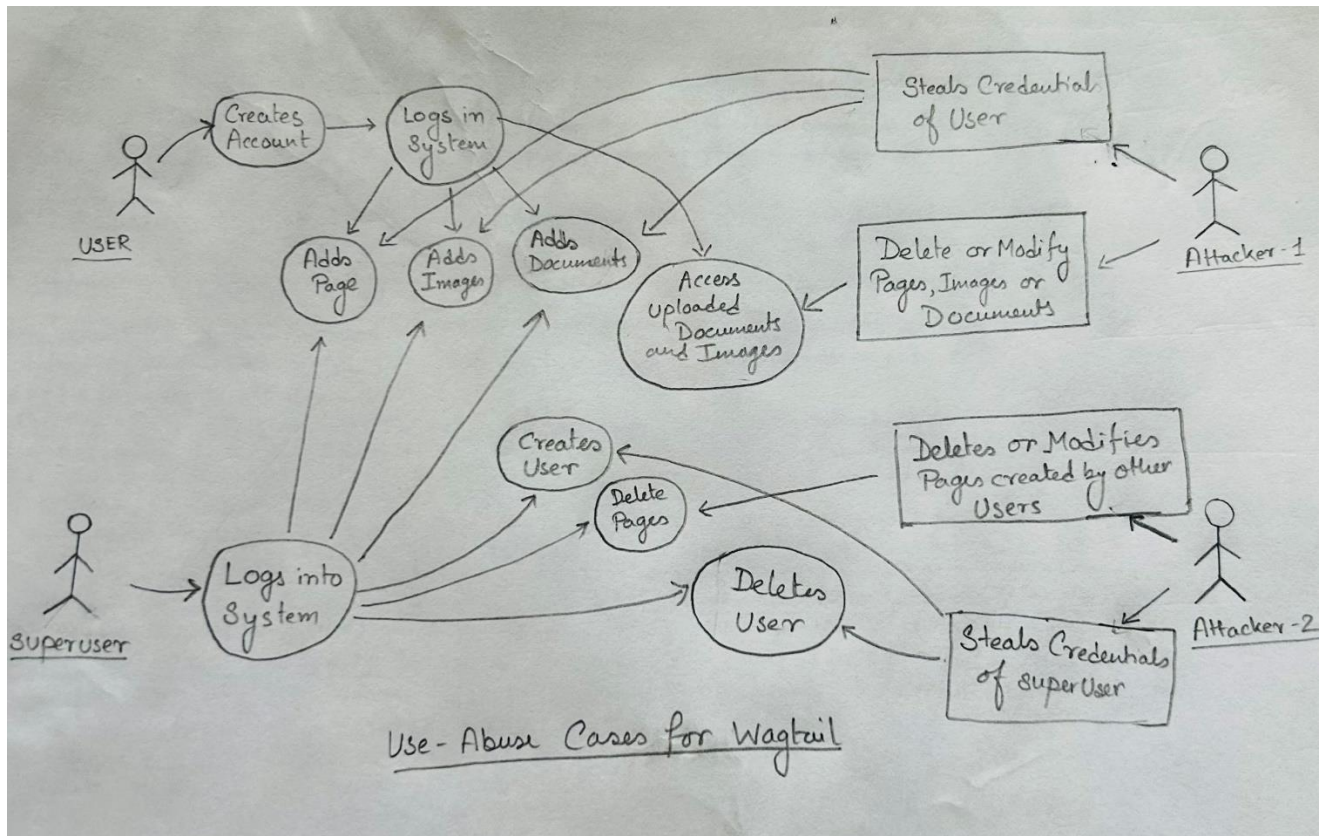
Objective: Authentication and Access Control - Prevent unauthorized access to user accounts and sensitive information.

10. Requirement: Error messages must be limited to prevent leakage of sensitive information (e.g. database errors or stack trace).

Objective: Confidentiality - Prevent unauthorized access to sensitive information through error messages.

Part 4: Abuse / Misuse Diagram

Create an abuse/misuse diagram. Your diagram should have at least 2 attackers and four possible attacks. You must also complete the template presented in the requirements lecture for 1 of the attacks.



Detailed misuse case

- **Summary:** A hacker attempts to gain unauthorized access to the Wagtail application by exploiting vulnerabilities or using stolen credentials.
- **Basic Path:**
 - a. The hacker identifies a vulnerability in the application, such as an unpatched security flaw.
 - b. The hacker uses the vulnerability to gain access to the application or user data.
- **Alternative Paths:**
 - a. If the hacker is unable to identify a vulnerability, they may attempt to brute force passwords or use other methods to gain access.
 - b. If the hacker is unable to gain access to the application, they may attempt to access the system through a third-party application or service that has been integrated with Wagtail.

- **Capture Points:** *a.* The point at which the hacker gains unauthorized access to the application or user data.

b. Any attempts by the hacker to exploit vulnerabilities or brute force passwords.

- **Preconditions:** *a.* The application is accessible via the internet. *b.* The hacker has the necessary knowledge and tools to exploit vulnerabilities or use stolen credentials.
- **Assumptions:** *a.* The application has not been properly secured or patched. *b.* The hacker can evade any security controls or measures put in place by the organization.
- **Worst case threat:** The hacker gains access to sensitive user data or other confidential information, which could result in financial loss, reputational damage, or legal consequences.
- **Capture guarantee:** The system should capture any attempts by the hacker to exploit vulnerabilities or gain unauthorized access to the application.
- **Related business rules:** *a.* All user accounts must be secured with strong passwords and two-factor authentication.

b. All third-party integrations must be thoroughly vetted and secured before being allowed to access the system.

- **Potential misuse profile:** *a.* The hacker is highly skilled in exploiting vulnerabilities and has access to advanced hacking tools.

b. The hacker has a motive for gaining unauthorized access, such as financial gain or malicious intent.

- **Stakeholders and threats:** *a.* Stakeholders: The organization, users of the application, regulatory bodies.

b. Threats: Unauthorized access to user data, financial loss, reputational damage, legal consequences.

Project: Wagtail

([Repo Link](#))

Members:

Suparno Saha, ssaha7

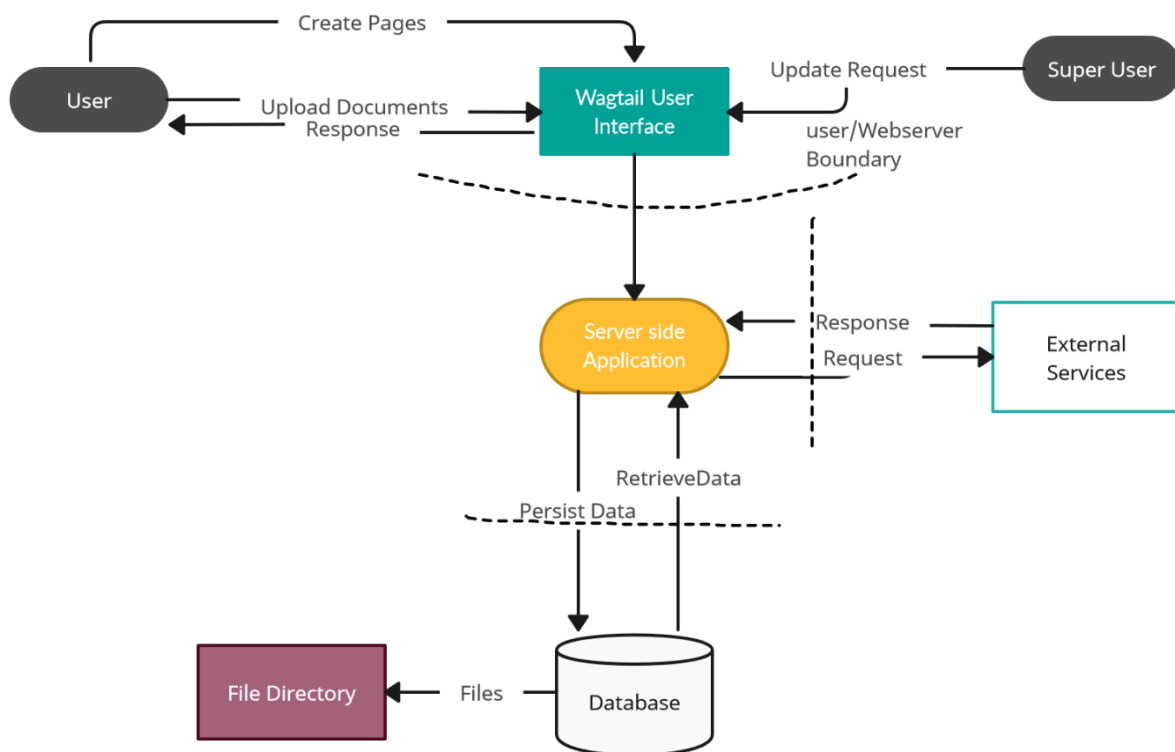
Daksh Mehta, dmehta4

Rohan Shiveshwarkar, rsshives

Phase 2 Report

Part 1: STRIDE Threat Analysis

Create a data-flow diagram for your system. Create a list of at least 10 threats identified using STRIDE. Each threat should include the threat category, threat description, mitigation description, and DREAD score.



Dataflow Diagram of WagTail

Here's a list of 10 potential threats for a Wagtail application, identified using the STRIDE model, along with their DREAD score breakdown:

1. **Spoofing:**

- Description: An attacker creates a fake user account and gains unauthorized access to the system. Through the fake user, the attacker may use cross-site scripting by making some changes to the view of the User application and make him/her click on links which may lead to passing URL path of the documents the user has uploaded in this application.
- Mitigation: Implement user authentication mechanisms like multi-factor authentication and use strong password policies.

DREAD Score Breakdown:

- Damage: 7
- Reproducibility: 6
- Exploitability: 5
- Affected Users: 7
- Discoverability: 6
- Total: 6.2

2. **Tampering:**

- Description: An attacker modifies data on the server or client-side. This can be done if a user's document which is being uploaded is being modified by the attacker. Being an application for managing contents of the users, data tampering is one of the greatest potential threats.
- Mitigation: Use HTTPS, implement data validation on the server and client-side which can be done by implementing static analysis tool, and implement integrity checks.

DREAD Score Breakdown:

- Damage: 8
- Reproducibility: 5
- Exploitability: 6
- Affected Users: 7
- Discoverability: 6
- Total: 6.4

3. **Repudiation:**

- Description: An attacker denies performing a specific action. This can happen if attacker enters in the application using credentials of other users (Identity Theft) and make changes in his/her contents and later on the user denies performing such actions.
- Mitigation: Implement secure logging, use digital signatures, and audit logs.

DREAD Score Breakdown:

- Damage: 4
- Reproducibility: 4
- Exploitability: 4
- Affected Users: 4
- Discoverability: 4
- Total: 4.0

4. **Information Disclosure:**

- Description: Sensitive information is leaked to unauthorized parties. This can happen if attacker gains access to the superuser credentials and make some of the documents uploaded by one user accessible to other users (Preferably himself/herself) and later access those data from his/her own account.
- Mitigation: Implement access controls, encrypt sensitive data, and use secure communication protocols.

DREAD Score Breakdown:

- Damage: 6
- Reproducibility: 5
- Exploitability: 6
- Affected Users: 6
- Discoverability: 6
- Total: 5.8

5. **Denial of Service (DoS):**

- Description: An attacker floods the system with traffic to make it unavailable to users. Though this can be avoided by using pay as you go Cloud servers, but again this is one of the easiest potential attacks which can be done to this application. The server can crash making user data unaccessable.
- Mitigation: Implement rate limiting, use a content delivery network (CDN), and use distributed denial of service (DDoS) mitigation services.

DREAD Score Breakdown:

- Damage: 8
- Reproducibility: 5
- Exploitability: 6
- Affected Users: 9
- Discoverability: 5
- Total: 6.6

6. **Elevation of Privilege:**

- Description: An attacker gains elevated access privileges to the system. This can be done by making a user a superuser and gaining access to other pages and

documents uploaded by other users. This may also be used to create denial of service by deleting a user account.

- Mitigation: Implement access controls, use least privilege principles, and implement secure coding practices.

DREAD Score Breakdown:

- Damage: 7
- Reproducibility: 5
- Exploitability: 6
- Affected Users: 5
- Discoverability: 5
- Total: 5.6

7. Cross-Site Scripting (XSS):

- Description: An attacker injects malicious scripts into a website, which is executed in a user's browser. This can lead to attacks through which attacker can gain access to documents and pages designed by the user. Moreover the attacker won't require to get access to the credentials to do that.
- Mitigation: Implement input validation and output encoding, use HTTP-only and secure cookies, and implement content security policies.

DREAD Score Breakdown:

- Damage: 6
- Reproducibility: 5
- Exploitability: 6
- Affected Users: 7
- Discoverability: 7
- Total: 6.2

8. Injection:

- Description: An attacker injects malicious code into a system, such as SQL injection, command injection, or XPath injection. Though we have tested this application for SQL injection attacks and couldn't find any such vulnerability, but there might be instances where it might still be vulnerable to SQL attacks.
- Mitigation: Implement input validation and output encoding, use prepared statements, and avoid concatenating user input with code.

DREAD Score Breakdown:

- Damage: 8
- Reproducibility: 4
- Exploitability: 4
- Affected Users: 7
- Discoverability: 5
- Total: 5.6

9. Broken Authentication and Session Management:

- Description: An attacker gains access to user accounts or session data by exploiting vulnerabilities in authentication and session management. This will not only provide privacy violation, but may also cause Availability and Confidentiality.
- Mitigation: Implement secure authentication mechanisms, use session timeouts, and use secure communication protocols.

DREAD Score Breakdown:

- Damage: 7
- Reproducibility: 4
- Exploitability: 6
- Affected Users: 6
- Discoverability: 4
- Total: 5.4

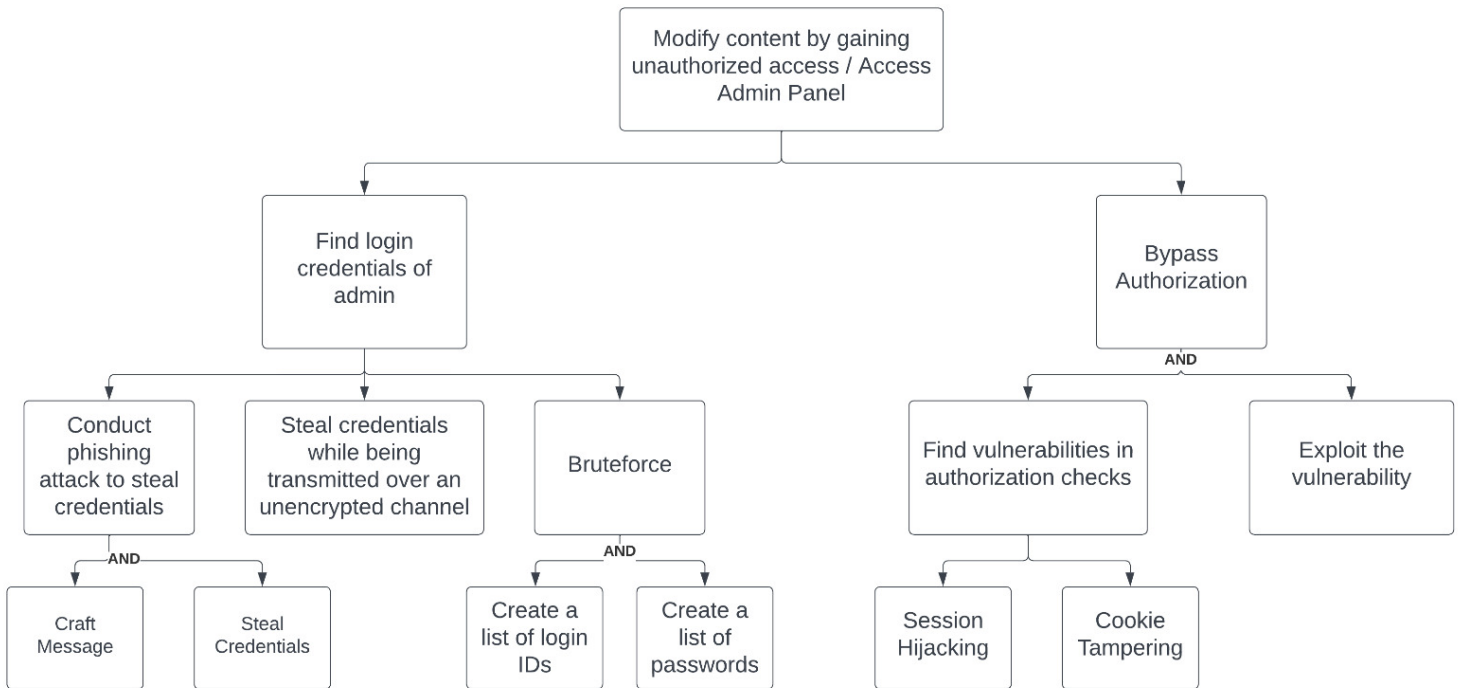
10. Security Misconfiguration:

- Description: An attacker exploits misconfigured security settings, such as default passwords, exposed sensitive information, or unsecured APIs.
- Mitigation: Implement secure configuration settings, remove default or unnecessary services, and implement secure coding practices.

DREAD Score Breakdown:

- Damage: 7
- Reproducibility: 4
- Exploitability: 6
- Affected Users: 7
- Discoverability: 4
- Total: 5.6

Part 2: Attack/Defense Trees



Part 3: Design Principles

Wagtail is a website used for authors to publish their content. It has a friendly user interface which can be designed as per the author's needs. It is scalable to millions of pages and thousands of editors. Editors can also embed images and videos. This brings along multiple ways to exploit the vulnerabilities in the website, and demotivate the intent of the website's usage.

A few important design principles described by Saltzer & Schroeder are a must-implemented:

1. Economy of mechanism

The Wagtail project offers a lot of freedom to the authors while publishing their content online and make it accessible to other authors worldwide. For this to work, the wagtail uses a lot of dependencies and packages which work together to make this happen. Many such packages have security vulnerabilities which can be exploited, risking the entire website. The design should be in such a way that the system uses a minimal set of dependencies and frameworks. Moreover, those should be kept up to date and should have a large userbase and is actively being updated with security patches.

2. Open Design

The Wagtail project has kept its codebase open to users in order to contribute and report of any security vulnerabilities discovered by the contributors. This follows the Open design principle of keeping the code open to scrutiny. The Wagtail project owners are open to knowing the security threats and attacks their website is susceptible to, and encourages sharing the knowledge.

3. Separation of privilege

This principle states that the access to resources should be divided among multiple parties to avoid critical damage to the system by one single entity. The access should be role based and each role should have access to only the required resources to perform their tasks. Here, let us take an example of an editor, another user (author) and an admin. The editor should have access to publish, edit and delete his post. The other user should only have access to that published post and not have access to the design page of the editor where the page can be edited or deleted. If for instance the editor published the post as anonymous, then the other user should not be able to view the editor's information. As for the admin, he/she can view, edit, delete anyone's post, view the account information of the editor but not be able to change their password, and send messages to the editor, which other users cannot.

4. Least privilege

This principle to some extent goes hand-in-hand with the previous principle. Here the idea is to give minimum access to regular users in order to use the website. In this case, the common jobs are publishing, editing, deleting their posts and viewing other posts published by other authors. Furthermore, they are only allowed to view their own sensitive account information and edit their credentials. Apart from that, the users should not have any other privileges such as viewing logs, seeing any other users' account activity, the unpublished work of other authors, etc.

5. Complete mediation

This principle states that the system should verify every access to resources by authenticating and authorizing all requests before granting access to resources. In this project, this is of utmost importance. Other users cannot edit other author's post by simply manipulating the URL by typing `www.wagtail.com/<post_id>/edit`. Same goes for viewing the author's sensitive information. If the admin is able to view the sensitive information and the attacker gets to know of the URL, even then the attacker should not be able to access it on his own, assuming he/she does not have access to the admin's account.

6. Psychological Acceptability

This principle states that the user should not get demotivated by the lengthy security features and not use the website at all. Security features are essential and should be implemented, but it should be with ease. Just because the process is easy does not mean that the security is superficial. In order to edit or delete a post, if the author has to every time login to their mail and click on a confirmation link to be redirected to another website and then be able to perform their desired tasks, then the author would not feel like using the website.

3 relevant design principles from the IEEE for this project are:

1. Confidentiality

This principle ensures that the sensitive information of the author such as address, contact information, birthdate, credentials, interests/preferences to posts should all be secured and properly encrypted. It should be prone to leaking of any sorts.

2. Availability

This principle states that the posts published by the authors should not be available in any form to other places such as other websites in sort of an extract or screenshot unless allowed by the user. This principle is held for the owners of the website, since the other users can post screenshots of the users post and cannot be monitored in case of this website. (This can be prevented by allowing a no screenshot policy enforced by Netflix)

3. Authentication

This principle states that the user performing any action should be properly authenticated and verify their credentials in forms such as passwords, multi factor authentication (such as Google Authenticator app which generates limited time codes) or biometrics (such as enabling Face-ID on the application). These are ways to properly authenticate the user and not be prone to impersonation/identity theft.

Project: Wagtail

([Repo Link](#))

Members:

Suparno Saha, ssaha7

Daksh Mehta, dmehta4

Rohan Shiveshwarkar, rsshives

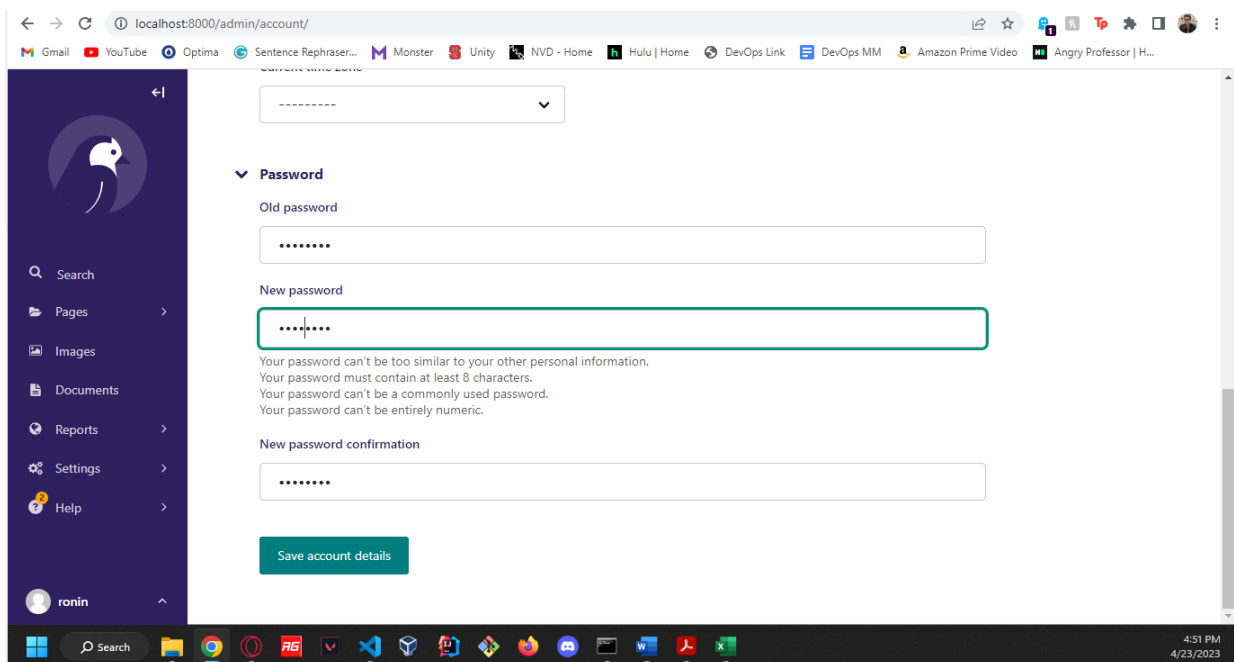
Phase 3 Report

Part 1: OWASP Application Security Verification Standard

The OWASP Application Security Verification Standard (ASVS) is a set of guidelines for verifying the security of web applications.

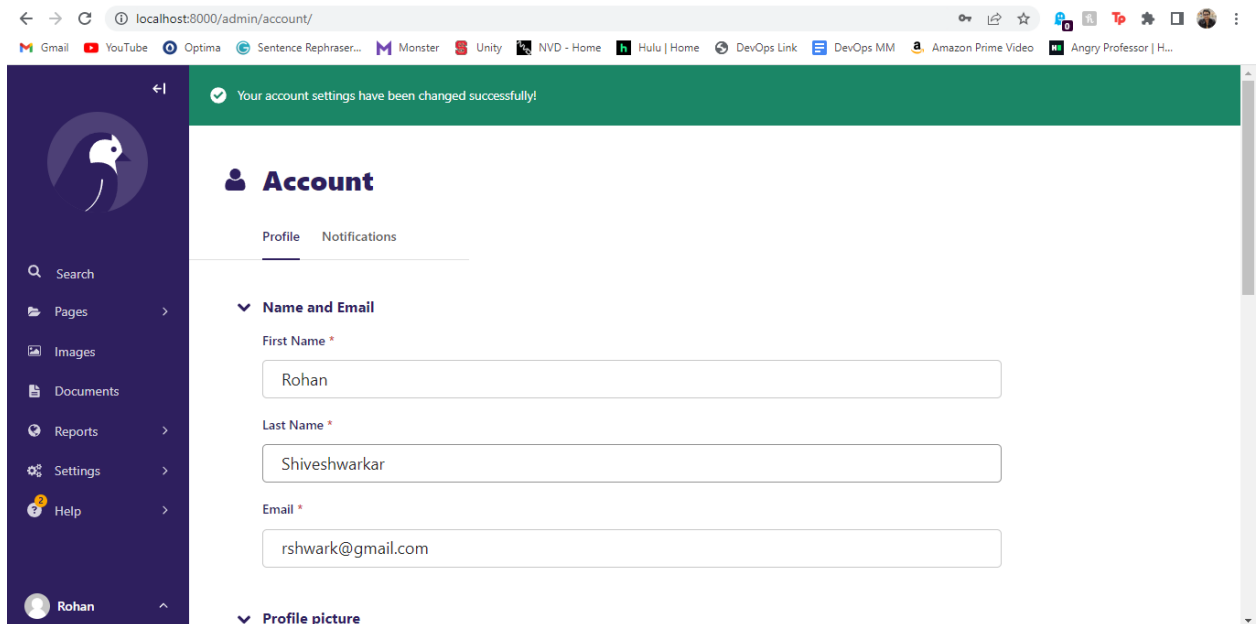
1. V2.1.1 - Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined)

For verifying this requirement, we attempt to change the password of an existing user. Ideally as per this requirement, the application is not to accept any passwords having length less than 12 characters. Although, in this application, the minimum length is set to 8 characters.



The screenshot shows a web browser window at the URL `localhost:8000/admin/account/`. The page has a dark purple sidebar on the left with a search icon and links for Pages, Images, Documents, Reports, Settings, and Help. The main content area is white and contains a 'Password' section with a dropdown menu, 'Old password', 'New password', and 'New password confirmation' fields. The 'New password' field is highlighted with a green border. Below the fields, there are four error messages: 'Your password can't be too similar to your other personal information.', 'Your password must contain at least 8 characters.', 'Your password can't be a commonly used password.', and 'Your password can't be entirely numeric.' A green 'Save account details' button is at the bottom. The browser's taskbar at the bottom shows various application icons and the system clock indicating 4:51 PM on 4/23/2023.

Here, we are attempting to change the password with a new password having length of 9 characters.



Here, we can see that the password of 9 characters in length has been accepted.

Thus, this requirement is not met for this application.

2. V3.1.1 - Verify the application never reveals session tokens in URL parameters.

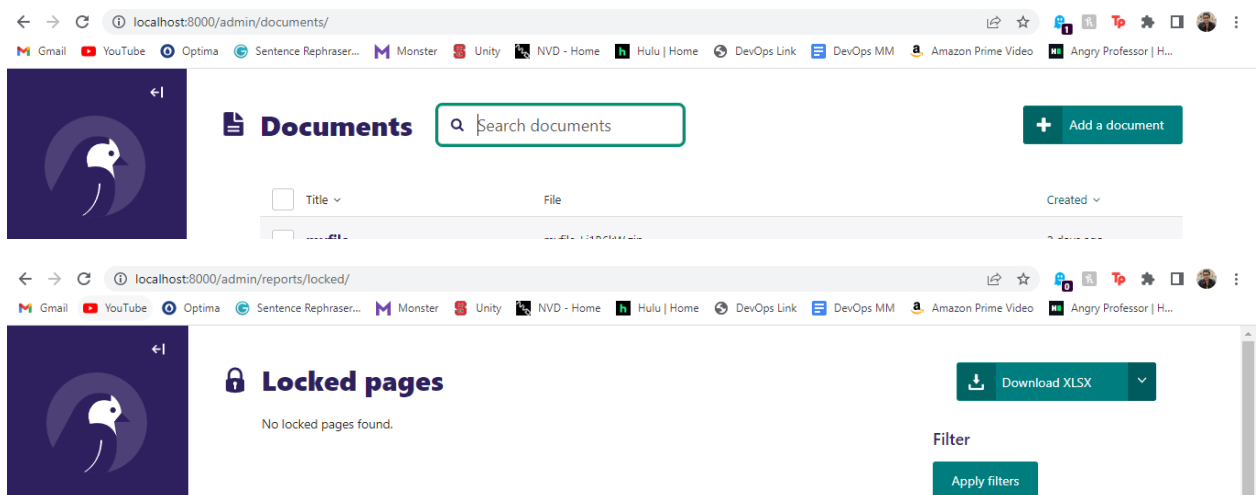
This requirement is to ensure that the application does not reveal the session token in the url parameters so as to avoid risks such as session hijacking, XSS attacks, and search engine indexing.

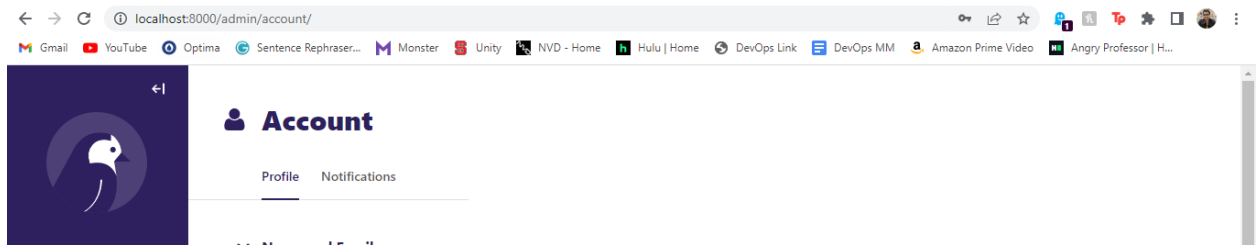
To test this requirement, we can analyse the codebase to see if the urls have sent the session token as a parameter.

We can analyse this from the admin/views/account.py file in the codebase. Here, the session token is being created and handled. From this file, we can conclude that the session token is not being passed as a url parameter.

Furthermore, we can manually test the application and look for the urls being generated.

For testing purposes, we are showing the important urls for the admin in this report.





These urls are only accessible to the admin. As we can see, the urls do not have the session token as a parameter, nor can we see any trace of the session token in the Network tab while inspecting the application.

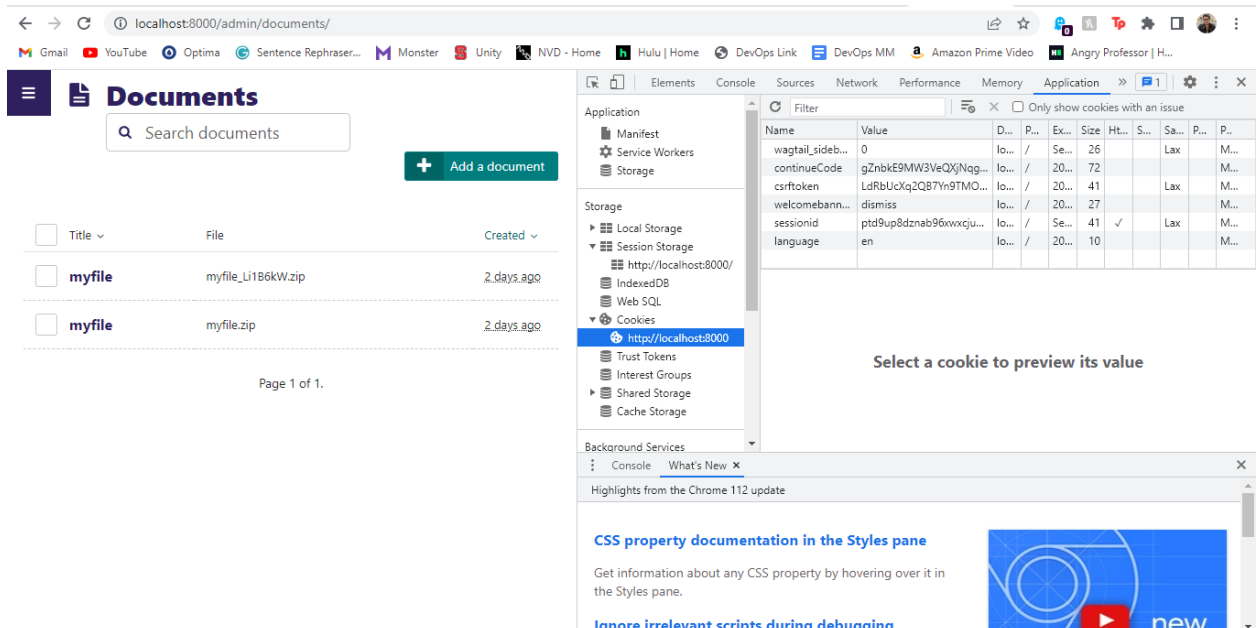
Thus, we can conclude that this requirement is completed in this application.

3. V3.3.1 - Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties

This requirement is to ensure that after a logged in user logs out of their account, the session token is invalidated and the user cannot presume activities which require a sign-in, after being logged out, and pressing the back button does not take the application back to the signed-in screen.

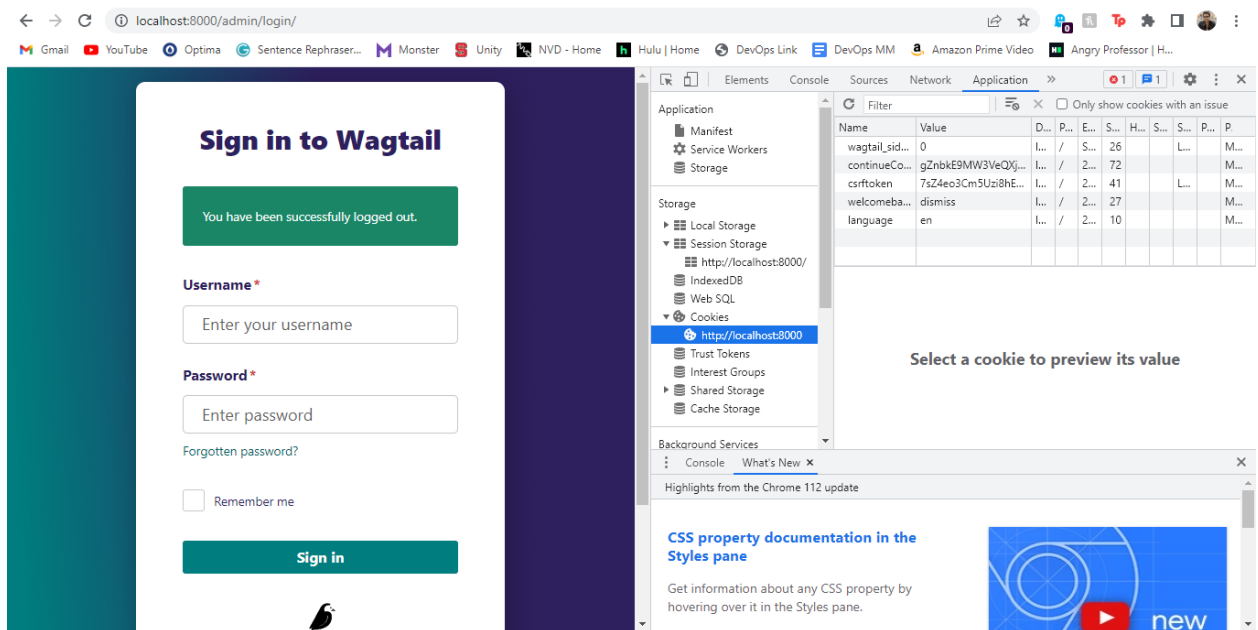
This requirement can be achieved by manually testing the application.

We are currently logged in as a user in the application.



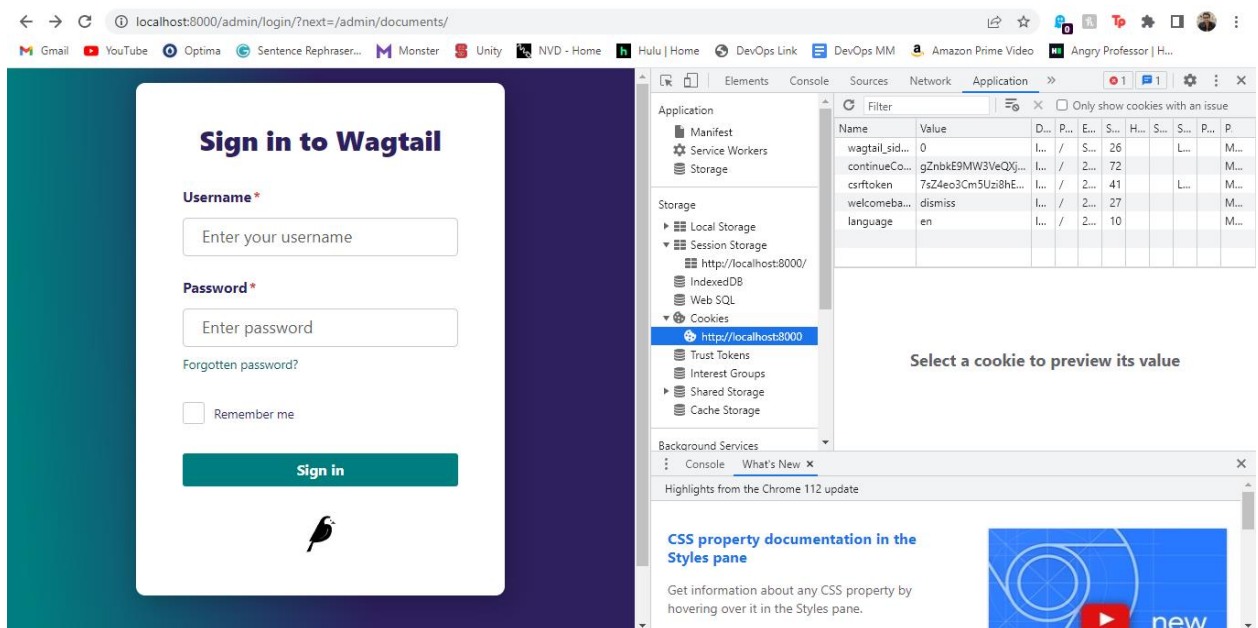
Here we can see the `sessionid` cookie, which is generated, indicating that the user is logged in.

Now, we logout of our account and are redirected to the sign in screen.



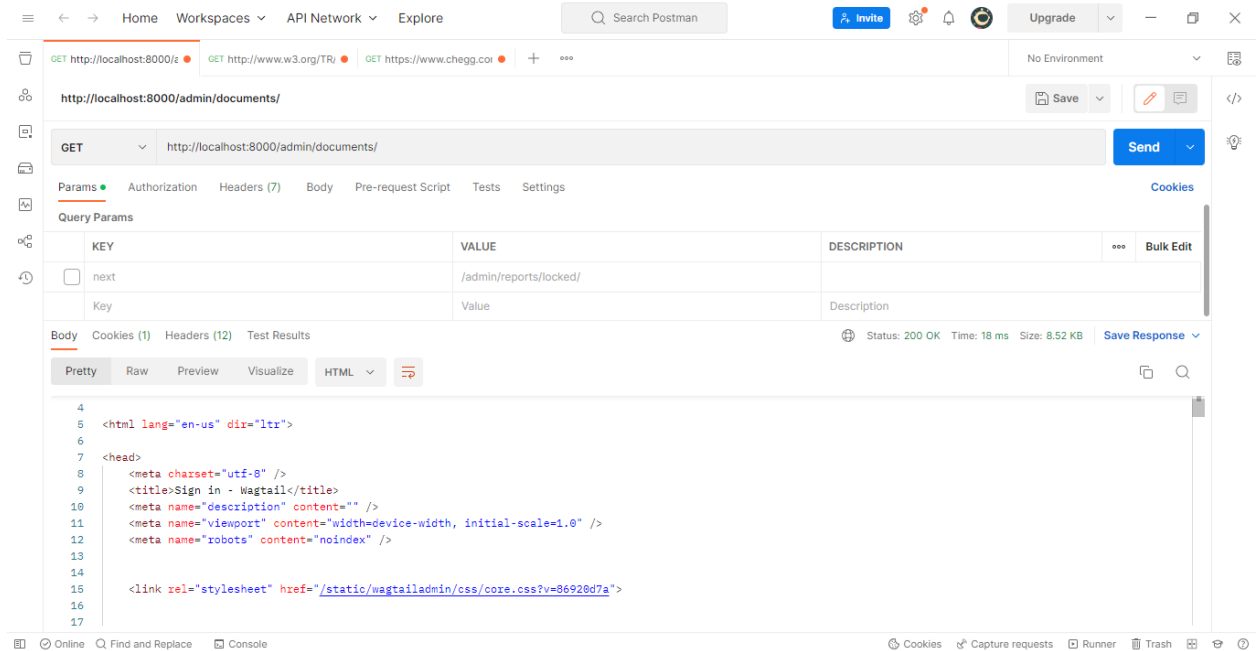
As we can see, the `sessionid` cookie is deleted.

Now we try to press the back button.

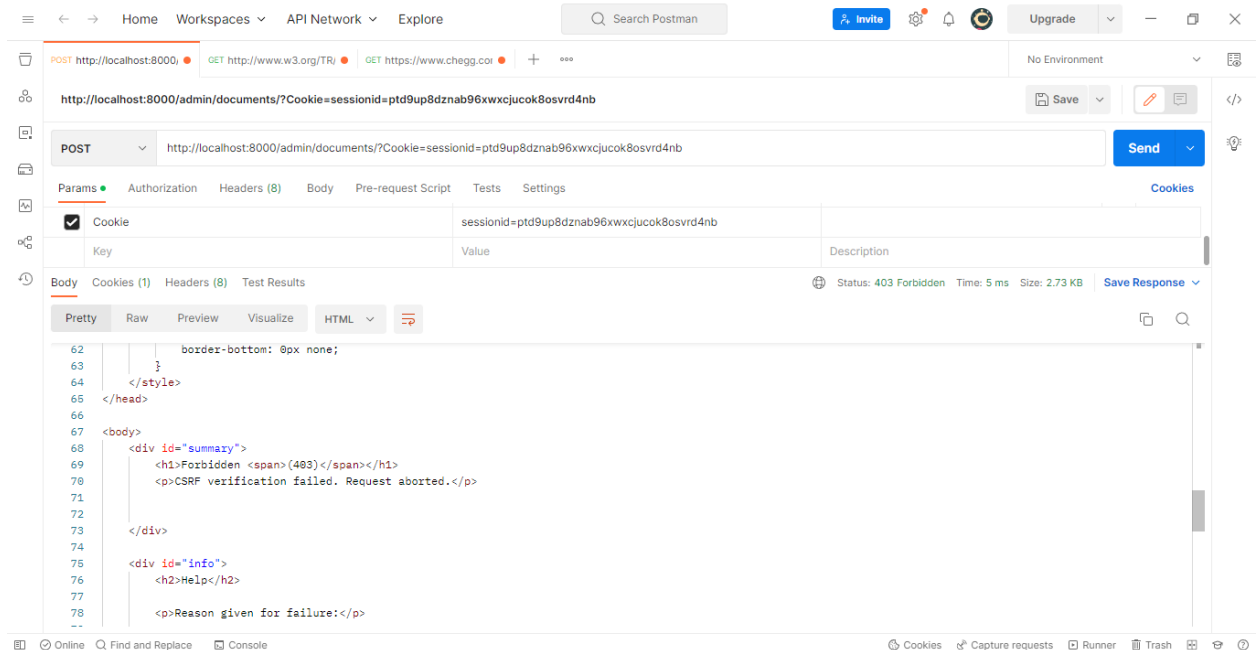


On doing so, we are again redirected to the login screen. Although, a side note to be considered, is that the previous activities' urls can be seen in the browser, and a hacker can learn of the various urls used in the application.

Even in Postman, since we do not have the cookie: `sessionid`, we are being provided the sign-in screen.



After adding the cookie value of `sessionid` received from the browser login to Postman, we are still unable to access the data of the signed in user.



Thus, we can conclude that this requirement is satisfied, and the session tokens are invalidated after logout.

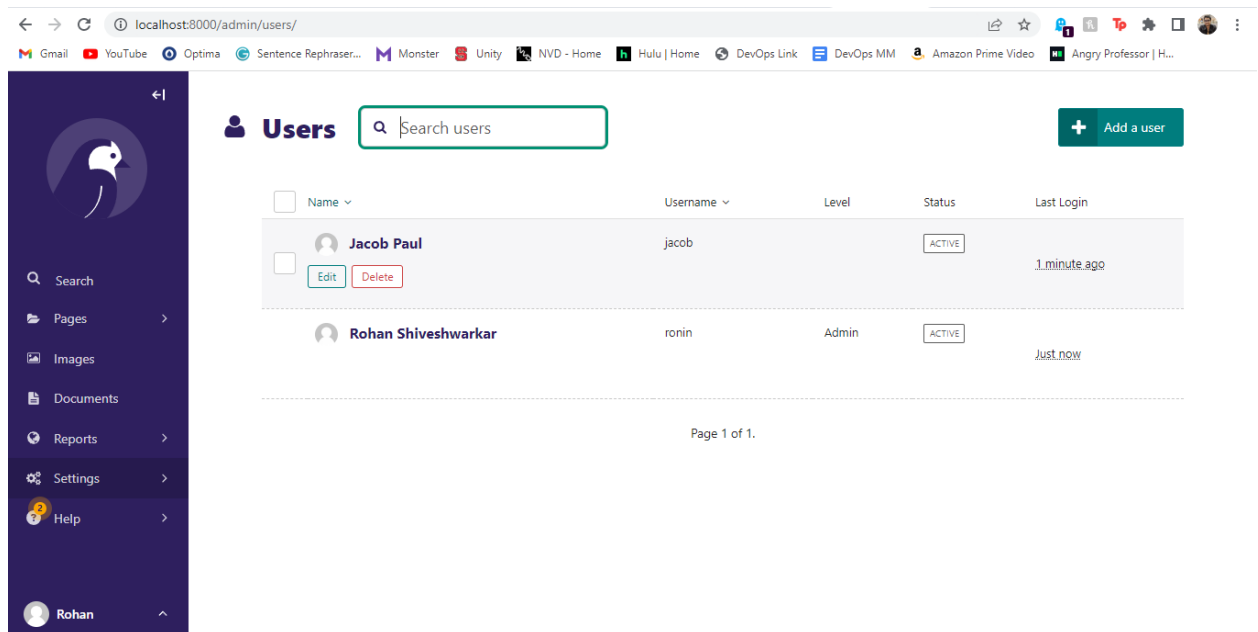
4. V4.1.1 - Verify that the application enforces access control rules on a trusted service layer, especially if client-side access control is present and could be bypassed.

This requirement is to ensure that the access control rules are followed, and that users not having access to certain resources cannot simply access them by altering the url.

For testing this requirement, we can manually test it at client side.

Currently we have 2 users – ronin (admin level access) and jacob (simple editor access)

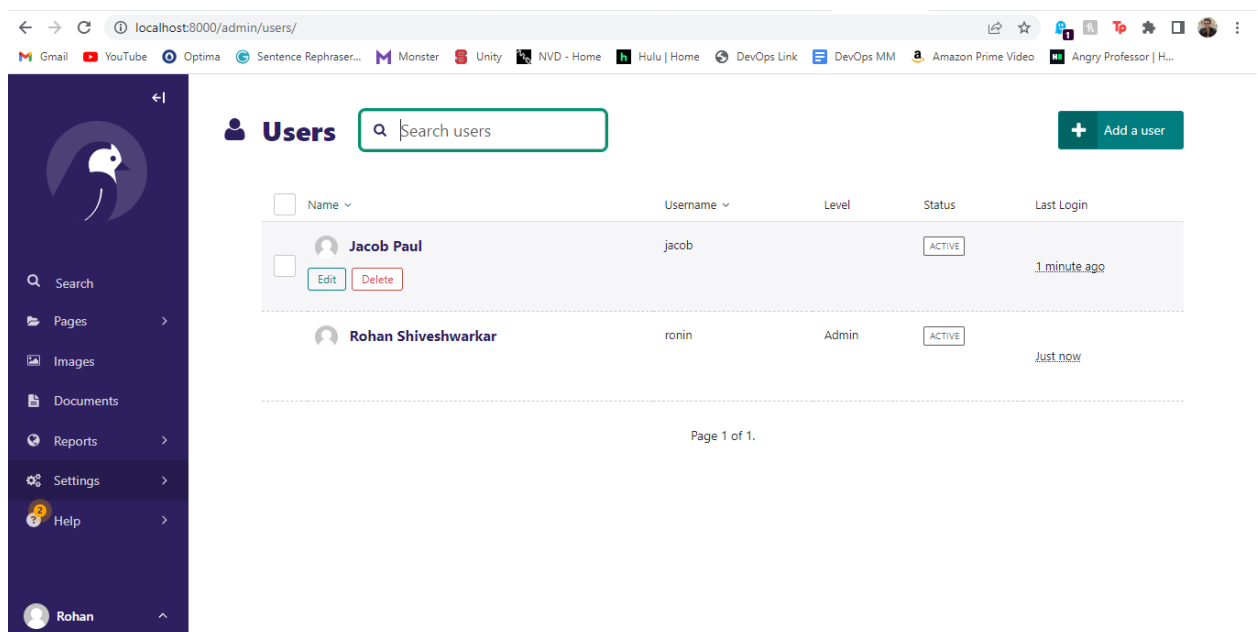
Verify that the application enforces access control rules on a trusted service layer, especially if client-side access control is present and could be bypassed.



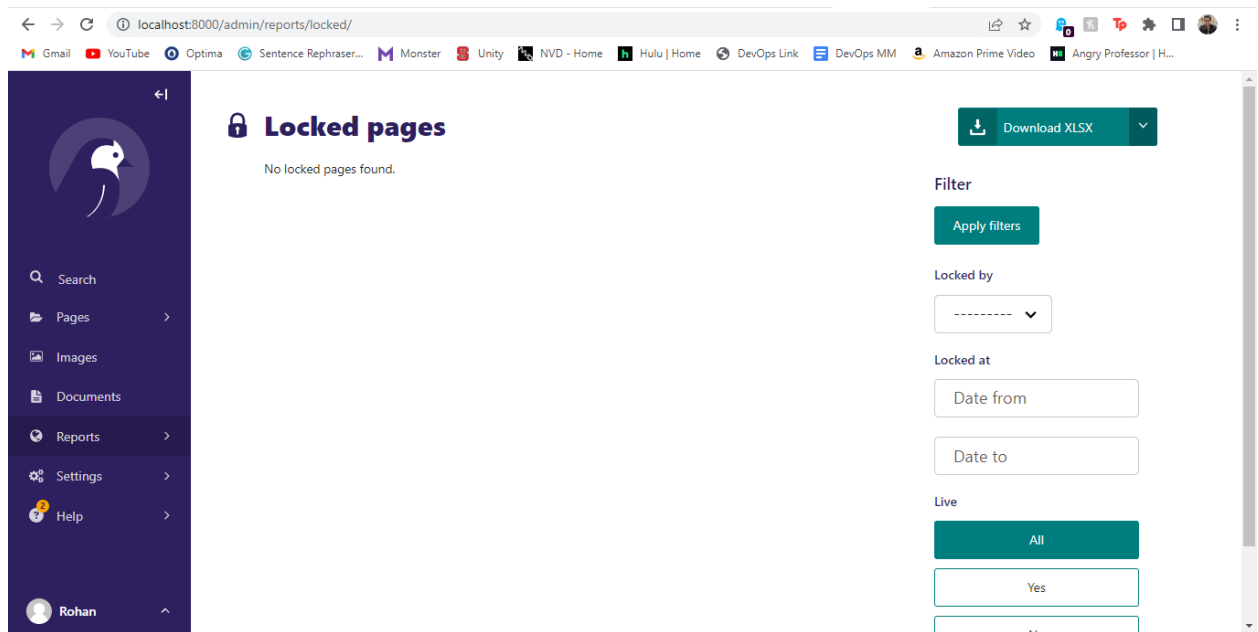
Now, on the ronin (admin) account, we can access admin-only resources such as /users, /reports/locked, etc.

As shown below, we can access the following resources on the admin account:

/users url

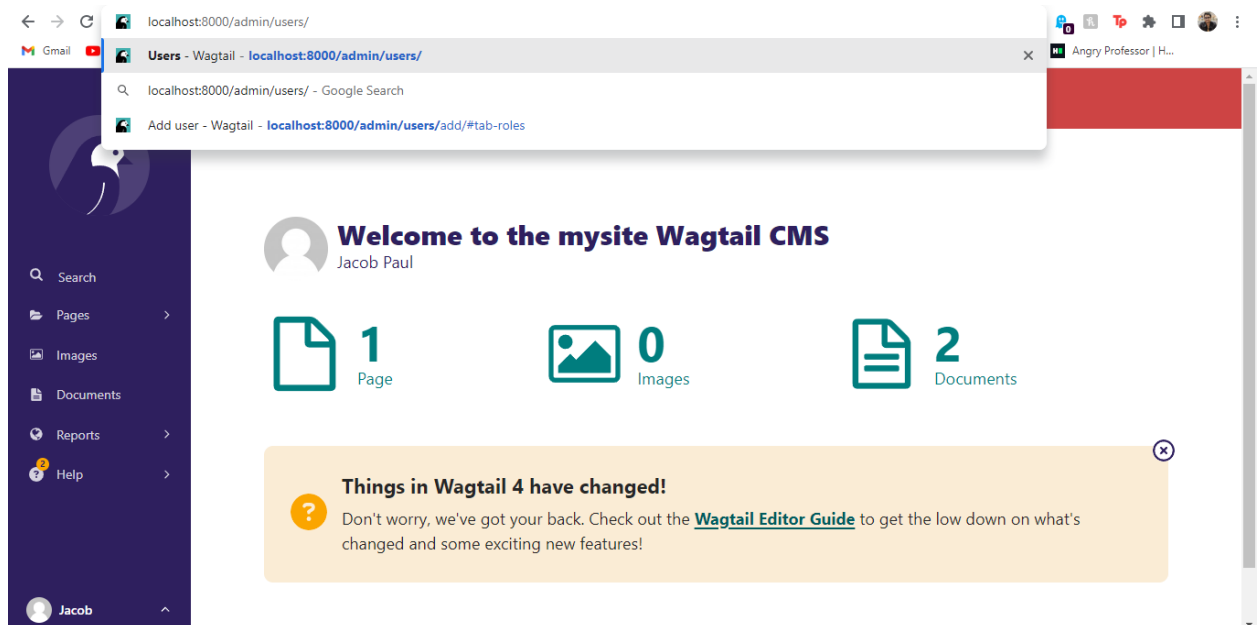


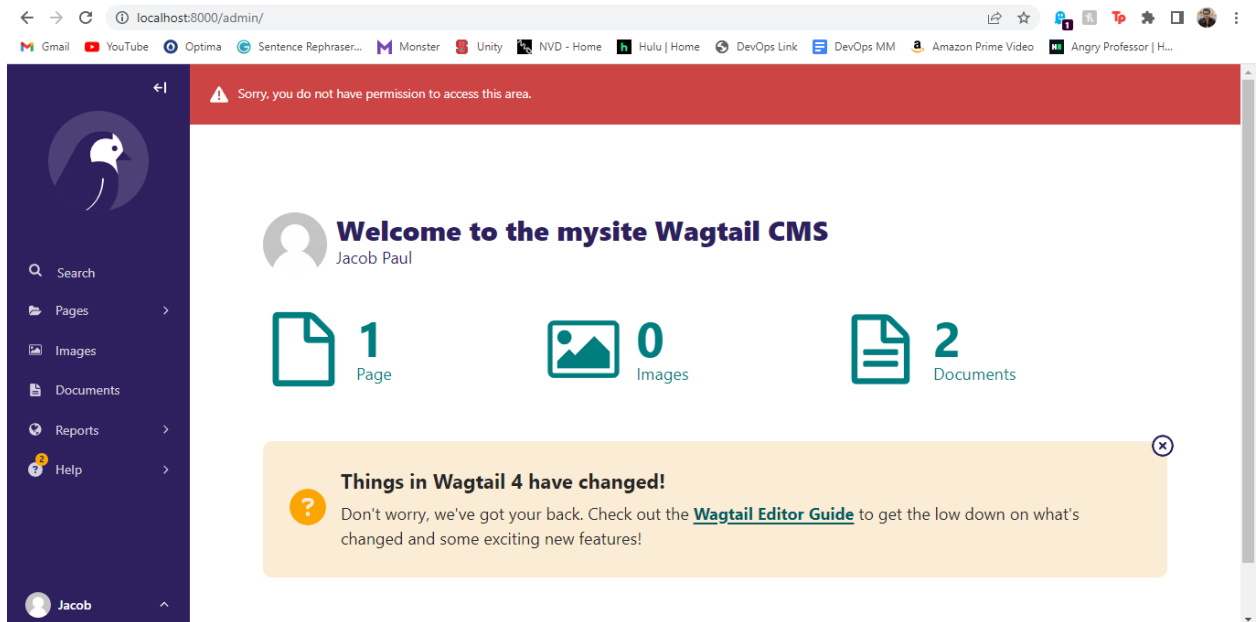
/reports/locked url



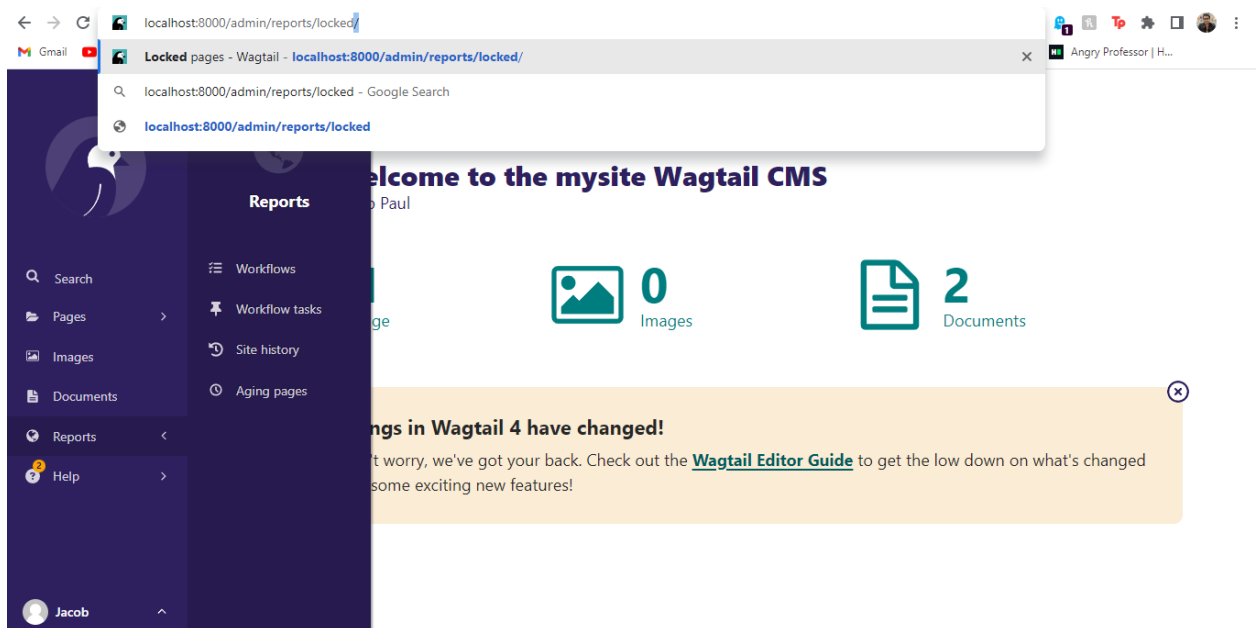
But after logging in onto the Jacob account (basic access), we get the following errors.

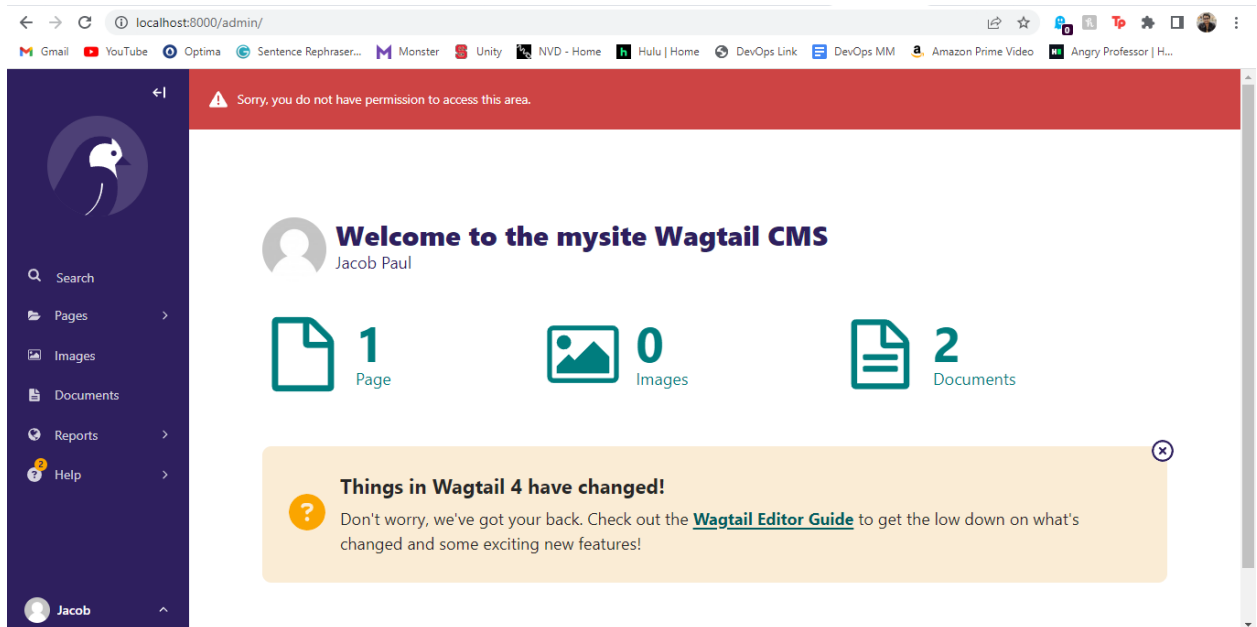
For /users





For `/reports/locked`





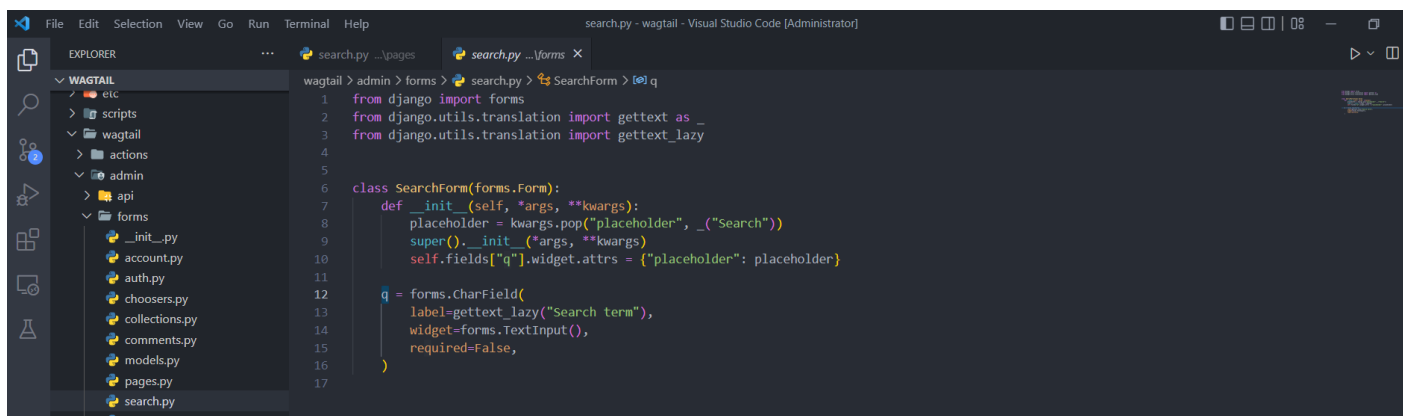
This shows that this requirement is fulfilled.

5. V5.1.3 - Verify that all input (HTML form fields, REST requests, URL parameters, HTTP headers, cookies, batch files, RSS feeds, etc) is validated using positive validation (allow lists).

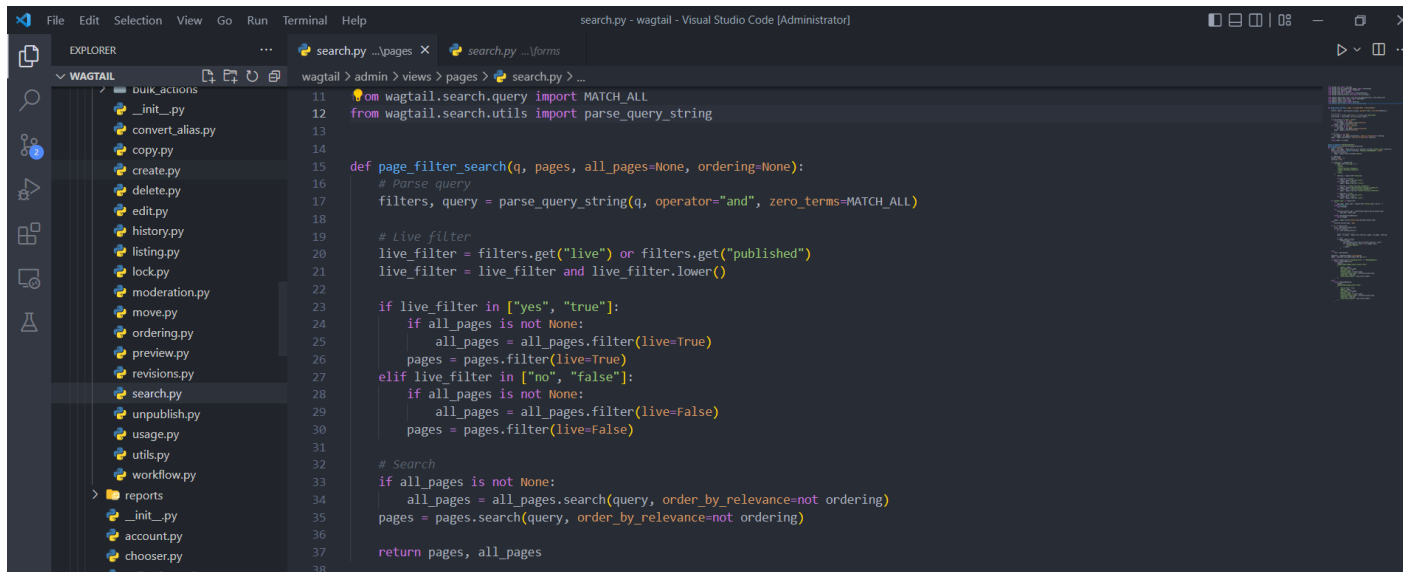
This requirement is to ensure that the application is not prone Injection attacks, DoS attacks, etc.

For testing this requirement, we can analyse the codebase. Here, we are considering the search functionality in the application.

The application is accepting all the form data in a 'q' variable in `/admin/forms/search.py` in the codebase as shown in the figure below.



This **'q'** variable is further passed on to `/admin/views/pages/search.py` where the search functionality is implemented.



```
11 from wagtail.search.query import MATCH_ALL
12 from wagtail.search.utils import parse_query_string
13
14
15 def page_filter_search(q, pages, all_pages=None, ordering=None):
16     # Parse query
17     filters, query = parse_query_string(q, operator="and", zero_terms=MATCH_ALL)
18
19     # Live filter
20     live_filter = filters.get("live") or filters.get("published")
21     live_filter = live_filter and live_filter.lower()
22
23     if live_filter in ["yes", "true"]:
24         if all_pages is not None:
25             all_pages = all_pages.filter(live=True)
26         pages = pages.filter(live=True)
27     elif live_filter in ["no", "false"]:
28         if all_pages is not None:
29             all_pages = all_pages.filter(live=False)
30         pages = pages.filter(live=False)
31
32     # Search
33     if all_pages is not None:
34         all_pages = all_pages.search(query, order_by_relevance=not ordering)
35     pages = pages.search(query, order_by_relevance=not ordering)
36
37     return pages, all_pages
38
```

As we can see, the variable **'q'** is directly being passed to search for the matching content and the pages against the input is being returned. Here, no validation for user input using positive validation is being done. Similarly, in the entire project, no sort of positive validation is being done.

Thus, this requirement is not being fulfilled by this web application.

6. V6.1.2 -This version requires that the application enforces password complexity requirements when passwords are created or changed, such as minimum length, character complexity, and prohibited passwords.

To verify this requirement for an application, we would attempt to create a new account and set a password that violates the password complexity requirements, such as setting a password that is too short or lacks complexity. The application should then reject the password and prompt me to choose a new one that meets the requirements.

Password *

Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation *

As seen on the above screenshot, Wagtail has provided Password requirements which matches the requirements specified by OWASP. Moreover, the application checks for weak passwords and notify user to change them if they find the password to be too common irrespective of fulfilling the requirement.

Password *

Your password can't be too similar to your other personal information.
 Your password must contain at least 8 characters.
 Your password can't be a commonly used password.
 Your password can't be entirely numeric.

Password confirmation *

▲ This password is too common.

As the application enforces the password complexity requirements as specified by the ASVS, it meets this requirement.

7. V6.2.2 - Verify that the application enforces secure password storage, such as hashing and salting, and uses a sufficiently strong algorithm.

To check this requirement, we require to examine the database schema to determine how passwords are stored. We could also attempt to retrieve a password hash and verify that it is properly hashed and salted using a strong algorithm such as bcrypt or scrypt.

We found the passwords to be using encryption provided by built-in Django library for this project.

We tried to access the database of the system using pgAdmin4, which runs PostgreSQL database and found the table named “auth_user” having hashed password of all the users.

| | id [PK] Integer | password character varying (128) | last_login timestamp with time zone | is_superuser boolean | username character varying (150) |
|---|--------------------|---|--|-------------------------|-------------------------------------|
| 1 | 1 | pbkdf2_sha256\$390000\$F8RuVE2L34efjqAcshqLMV\$yVZpF2PtXFg9X5nFMUqu1qgmrbZqOGxUwf/qptmp5po= | 2023-04-21 23:37:48.732083-04 | true | supar |
| 2 | 2 | pbkdf2_sha256\$390000\$XwOCdWloOKpNrfenaWEcaG\$sn6gadEUO+y1n1a4kzoYIIULETIPbXlaMaiUngKbPr8= | 2023-04-21 23:37:26.273136-04 | false | suparno |
| 3 | 3 | pbkdf2_sha256\$390000\$XwOCdWloOKpNrfenaWEcaG\$sn6gadEUO+y1n1a4kzoYIIULETIPbXlaMaiUngKbPr8= | 2023-03-29 00:04:18.502859-04 | true | ss |
| 4 | 4 | pbkdf2_sha256\$390000\$NzhMVVtcReSYPyKAFmVYvH\$dNkhU6r++J7593wcEJP0r2AOJ2vNuUyYauPF8AVpo... | [null] | true | Sup |

On verifying we found the passwords to be properly hashed and could not be decrypted using any online decryptor.

Moreover, we tried copying encrypted password of one user and pasting it in to the password column of another user. But still the actual password of the first user is not working for the user whose password has been changed.

Hence, we can conclude that this system follows this OWASP verification standard.

8. V7.1.3 - Verify that the application enforces access controls on all resources based on the principle of least privilege.

To check this requirement, we would attempt to access resources that should only be accessible to authenticated users or specific roles, and verify that the system properly restricts us from accessing them if we do not have the required privileges.

We tried accessing the create user functionality using a normal user (without admin access) and couldn't perform those operations. Moreover, a normal user cannot access pages created by other users or documents uploaded by other users.

The below screenshots (Figure 1) show the access which are provided to admin user and (Figure 2) the access which are provided for other normal users.

Hence we can say that V7.1.3 of OWASP verification standard is followed by Wagtail.

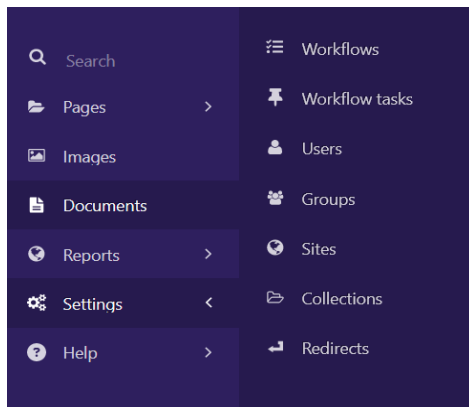


Figure 1

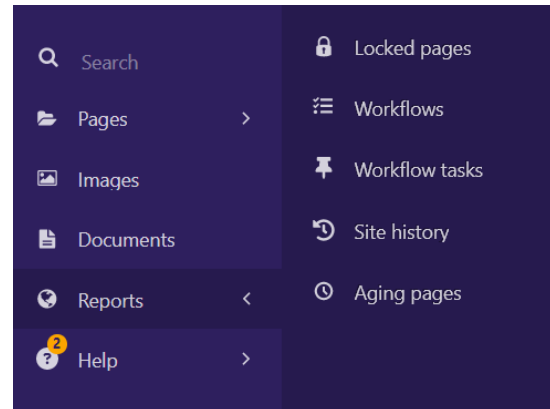
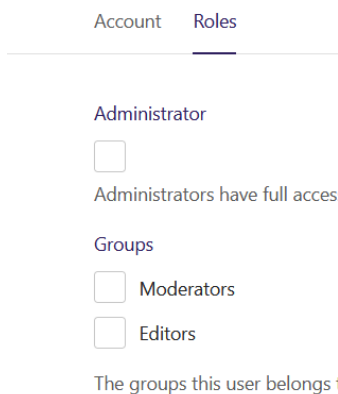


Figure 2

9. V7.1.4 - Verify that the application enforces access controls on all methods based on the principle of least privilege.

To check this requirement, we are required to examine the application code to determine whether access controls are applied consistently across all methods and endpoints. We would also attempt to call methods or access endpoints that should only be accessible to specific roles or users, and verify that it is properly restricted from accessing them if we do not have the required privileges.

Wagtail application has 3 set of roles, and their privileges are different, which follows the principle of least Privilege.



The administrator role has access to all other users, their pages and documents.

The second set of conditions provide access to add documents and pages uploaded by a group. Editor can edit pages and documents uploaded by other users within same group.

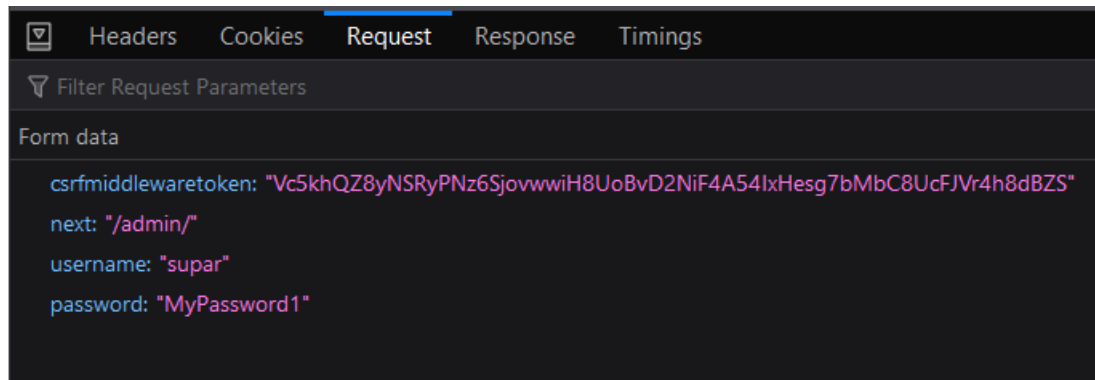
Moderators can only view the pages and documents added by users of the same group but couldn't edit them. No of these roles have access to other users' data.

Hence, we can say that Wagtail follows verification standard of OWASP V7.1.4.

10. V7.3.3 - Verify that the application protects sensitive data in transit using strong encryption, such as HTTPS/TLS with at least 128-bit encryption.

To check this requirement, we are required to examine the network traffic generated by the application using a network tab of Inspect element. We would verify that sensitive data such as authentication credentials and session tokens are transmitted over HTTPS/TLS with at least 128-bit encryption.

We tried reading the request and responses of the API from the “Networks” tab when login API is getting called.



As can be seen in the screenshot above, the password is being sent as text, without any encryption. But again, the TLS encryption is added to the system which makes it difficult for the Attackers to access these data from intercepting the network.

Hence, we can conclude that though the application sends sensitive data as open text through network, but the network is being encrypted using TLS, which follows the OWASP standard validation.

Part 2: Usability

1. Visibility of system status -
Wagtail satisfies this heuristic. It provides users with visible feedback about the operation being performed by the system. For example, it displays messages such as article updated/posted when a new page is posted.
2. Match between system and the real world -
Wagtail satisfies this heuristic to a great degree. Familiar terminology is used such as blogs, posts, pages, images, categories, tagging posts. However, some of the terminology used does not directly matches the real-world use case, one such example being "streamfields" which basically refers to a page that is not structured.
3. User control and freedom -
This heuristic is satisfied by Wagtail as users have the option to undo/redo their actions. There is also a clear navigation interface in the system and the system notifies users with a confirmation message when something is being deleted by the user.
4. Consistency and standards -
Wagtail meets this heuristic since it follows the standard web application design and resembles other content management systems by its interface.
5. Error prevention -
Wagtails satisfies this heuristic as it provides detailed error messages to let the user know what they did wrong and also provides well labelled and appropriately placed buttons so that the user does not perform an unwanted action by error.
6. Recognition rather than recall -
Wagtail satisfies this heuristic. It provides appropriately coloured/shaped buttons and labels which helps the users easily locate the functionality they want. However, users will have to remember the location of some of the actions and how to navigate to them using the navigation bar.
7. Flexibility and efficiency of use-
Wagtail provides keyboard shortcuts which can be used by more experienced users to speed up their actions and make it an efficient process for them. Thus, it meets this heuristic.
8. Aesthetic and minimalist design -
Wagtail has a very minimalistic and clean design which is clutter free and appealing to the user's eye too. No unnecessary information is displayed where it isn't needed. Thus, it satisfies this heuristic.
9. Help users recognize, diagnose, and recover from errors -
Wagtail provides clear error messages about what went wrong and suggests the appropriate actions on how to recover from those errors, such as when a user forgets to fill a form field it points out what is missing. Thus, it meets this heuristic.
10. Help and documentation -
Wagtail documentation available online which can be easily accessed. It is well written and is easy to read and follow. It also has code snippets which explain step by step how a certain feature can be implemented. Thus, it meets this heuristic.

Warning message: "Sorry, upload failed. This file is too big. Maximum file size allowed is 10.0 MB."

NEAT: The NEAT mnemonic stands for Necessary, Explained, Actionable, and Tested.

Necessary - It is necessary to inform the user that the file they are trying to upload is too large and the process cannot proceed further without user intervention.

Explained – The warning message is explained in a clear and concise manner.

Actionable - It is actionable by informing the user of the maximum size limit and the user can then upload a file within that limit.

Tested - It is testable by checking if when the user tries uploading a smaller file they are successful or not and whether if they again upload a file greater than the file size limit they are shown the same warning. Thus, covering both benign and malicious scenarios.

Explanation:

SPRUCE: The SPRUCE mnemonic stands for Source, Process, Risk, Unique knowledge, Choices, and Evidence.

- Source: The warning message is generated by Wagtail itself which is a trusted source.
- Process: The warning message tells the user the maximum file size and tells them to upload a file with a smaller size.
- Risk: The warning message explains to the user that the file cannot be uploaded until and unless it is lesser than the maximum file size. If the user makes the wrong decision, then he won't be able to upload the file they want.
- Unique Knowledge user has: The warning message asks the user to upload an alternative file which satisfies the maximum file size and that can only be chosen by the user.
- Choices: The warning message presents the user with a clear and actionable choice, to upload a smaller file within the maximum allowed file size limit.
- Evidence: The warning message provides evidence to the user that the issue can be solved by uploading a file within the maximum allowed file size limit and the process can be completed.

Part 3: Testing

1. Test Case ID: ASVS V5.1.3 - Input Validation

CWE: 20 - Improper Input Validation

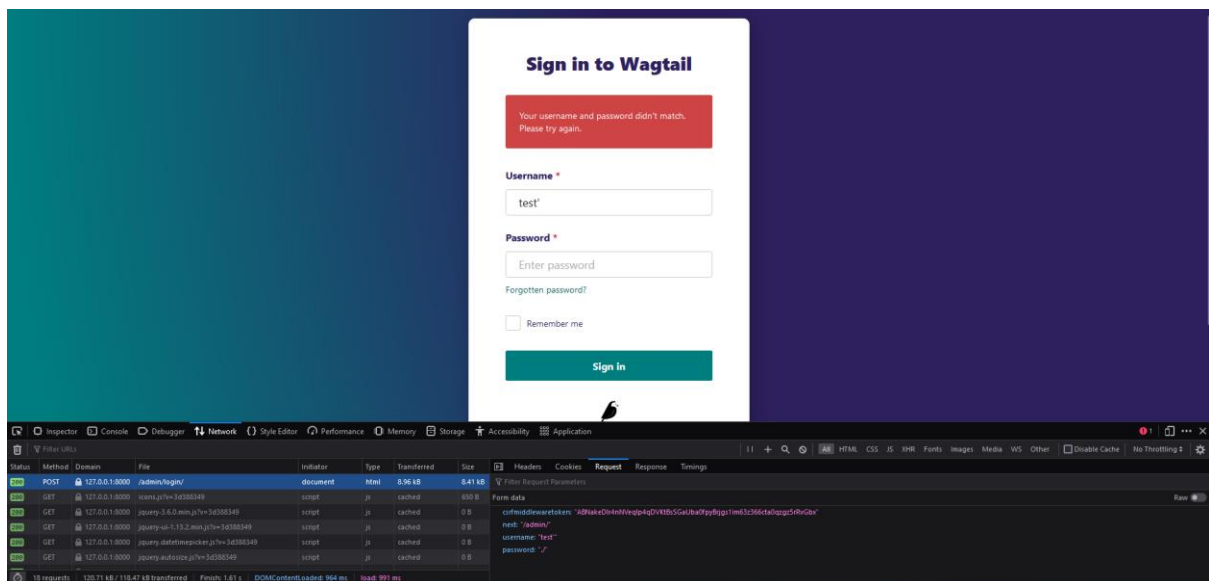
Description: Test case to verify that the system is checking for input validation and is sanitizing the input for special characters on the login page.

Steps:

1. Open wagtail and navigate to the login page.
2. In the field of username enter text that includes special characters such as " ' " or " < " or " > ".
3. Click on the sign in button and see if access is gained.
4. Check that the system has sanitized the inputted username and prevented any special characters from being inputted.

Expected Result: The system should prevent special characters from being entered in the username field and should result in an error and the user should not be able to gain access.

Actual Result:



2. Test Case ID: ASVS V2.2.1 - Authentication Testing

CWE: 307 - Improper Restriction of Excessive Authentication Attempts

Description: Test case to verify that the system does not allow brute force attacks to take place and blocks the user from performing too many login attempts within a short time window.

Steps:

1. Open wagtail and navigate to the admin login page.
2. Login using incorrect credentials multiple times within a few minutes (e.g., 5 times in a minute).
3. Check that the system locks the account and prevents further login attempts until the cooldown window is over.
4. Attempt to login again with the correct credentials. It should be accessible using the correct credentials.

Expected Result: The system should lock the account after a certain number of incorrect login attempts and after the cooldown window is over the account can be accessed again using the correct credentials.

Actual Result: Account is not locked even after numerous unsuccessful login attempts. This could lead to brute force attacks by attackers.

3. Test Case ID: ASVS V4.1.3 - Access Control

CWE-285: Improper Authorization

Description:

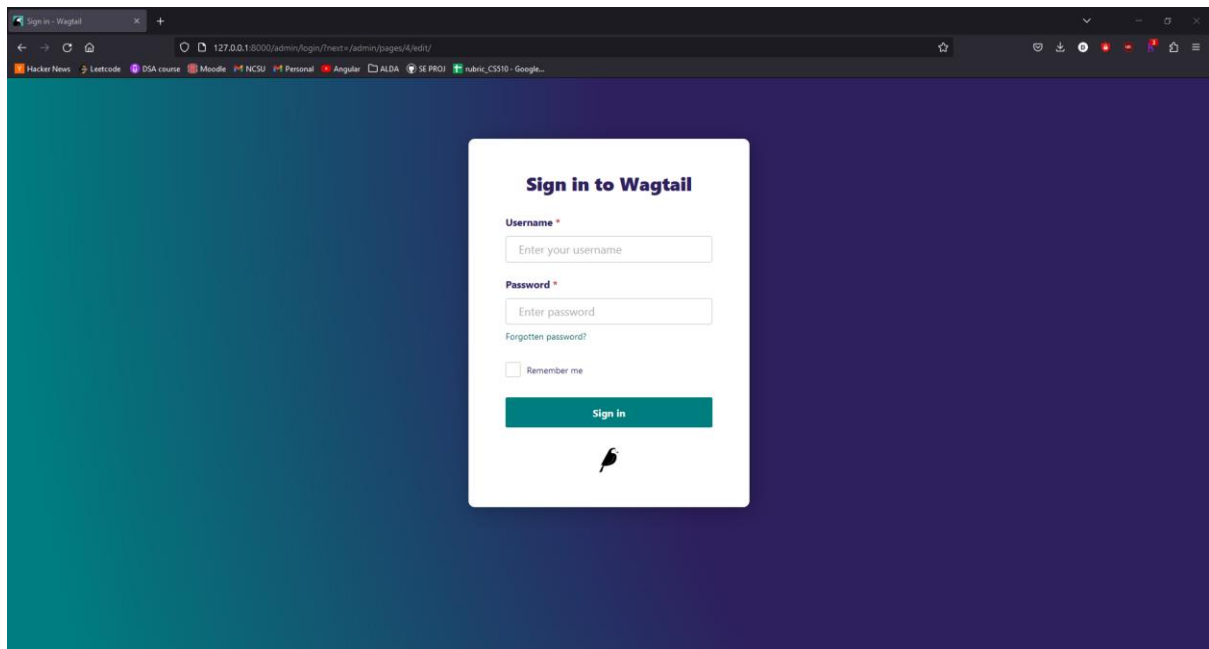
Test to verify that proper access controls have been implemented to restrict users from accessing sensitive data.

Steps:

1. Open wagtail and login using the administrator credentials and open the admin console.
2. Take a note of the URL and logout.
3. Attempt to access the same URL without logging in.
4. Verify that the system does not allow the user to access the sensitive data.
5. Verify that the user is redirected to the login page.

Expected Result: The unauthenticated users should not be able to access the sensitive data and should be redirected to the login page.

Actual Result: The user is redirected to the login page.



4. Test Case ID- ASVS V5.3.9 – Output Encoding and Injection Prevention

CWE: 20 – Javascript injection of Local File Inclusion or Remote File Inclusion attacks.

Description:

A JavaScript injection vulnerability is a subtype of cross site scripting (XSS) that involves the ability to inject arbitrary JavaScript code that is executed by the application inside the victim's browser. This vulnerability can have many consequences, like the disclosure of a user's session cookies that could be used to impersonate the victim, or, more generally, it can allow the attacker to modify the page content seen by the victims or the application's behavior. Attackers can perform this attacks either by including local files within system directory or extracting data from a malicious Remote File.

Steps-

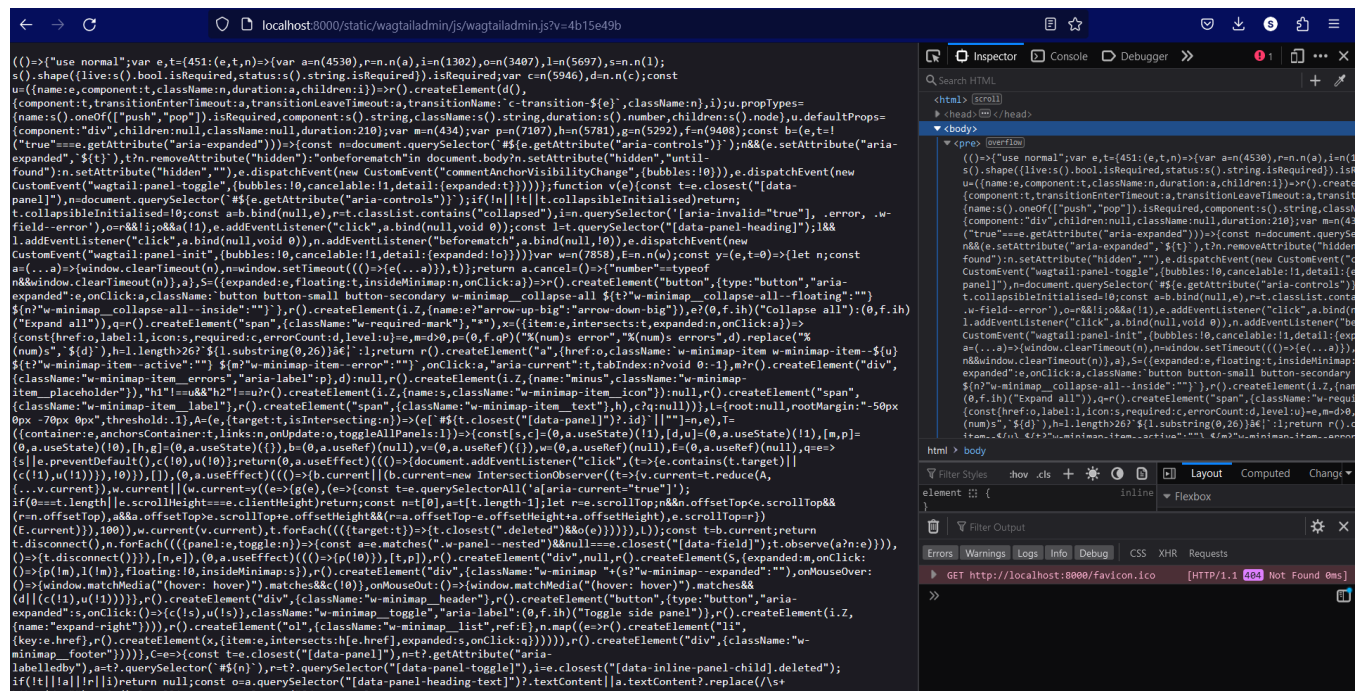
1. Login to the wagtail application as a user from the login page.
2. Right click and click on inspect element.
3. From the menu at the top select inspector.
4. Look for script tags with src attribute. Copy the link present inside the src attribute.
5. Open a new tab in browser and type- localhost:<port number>/<the url copied> and press enter.

Expected Result: The test should also verify that the web application provides appropriate error messages to the user when such attacks are attempted, rather than allowing the attack to succeed silently.

Additionally, the test should verify that the web application logs any attempted attacks and generates alerts or notifications to security personnel, allowing them to investigate and respond to the attack.

Actual Result:

You would be able to see from where the static data is getting fetched. The data can also be modified to use data from any external file by the attacker.



As shown in the screenshot attached above, this test reveals a lot of static data which which reveals a lot of information.

For example-

```
document.addEventListener("keydown", (e => { "Escape" === e.key && s() } ) )(), function() { const t = document.querySelector("[data-form-side]"); if (!e) return; const t = "formSideExplorer" in e.dataset, n = document.querySelector("[data-form-side-resize-grip]"), a = document.querySelector("[data-form-side-width-input]");
```

This code snippet reveals how the dataset is being managed to show it in the page when rendering in the DOM.

Though these couldn't be modified to persist in the server, which is hosting the application, but I feel that revelation of this data can lead to Cross Site Scripting attacks.

5. Test Case ID- ASVS V1.8.2 Data Protection and Privacy Architecture

CWE 319 - Verify that all protection levels have an associated set of protection requirements, such as encryption requirements.

Description: The purpose of this test is to ensure that all protection levels defined in the system have associated protection requirements, such as encryption requirements, integrity requirements, retention, privacy, and other confidentiality requirements. It also aims to verify that these protection requirements are appropriately applied in the architecture.

Steps:

1. Identify all the protection levels defined in the system architecture.

2. Review the system documentation and requirements to identify the protection requirements associated with each protection level.
3. Verify that each protection level has a set of associated protection requirements.
4. Verify that the protection requirements are appropriately applied in the architecture, including the use of encryption, access controls, and other security measures.
5. Test the protection measures by attempting to access or modify data that should be protected at each protection level, using appropriate test data and tools.
6. Verify that the protection measures prevent unauthorized access or modification of the data.
7. Verify that the protection measures are consistent with industry best practices and regulatory requirements.
8. If any protection requirements are missing or not properly implemented, report the issue to the development team and document the issue for follow-up testing.

Expected Result: The expected result of this test is that all protection levels defined in the system have associated protection requirements, and that these requirements are appropriately applied in the system architecture. All protection measures should be tested and found to prevent unauthorized access or modification of data, and should be consistent with industry best practices and regulatory requirements.

Actual Result: The project uses permission and authorization standards which comes built-in with Django. The authentication is done using the following code - [django.contrib.auth.authenticate\(\)](#).

We searched through the internet and found that the inbuilt authentication provided by Django is quite secure and protected. The protected Private pages were vulnerable to timing attacks on password which was addressed in release CVE-2020-11037.

6. TestCase ID – ASVS V7.4.2 Error Handling

CWE 544- Verify that exception handling (or a functional equivalent) is used across the codebase to account for expected and unexpected error conditions.

Description: The purpose of this black box testing is to verify that exception handling (or a functional equivalent) is used consistently across the codebase in the Wagtail project to account for both expected and unexpected error conditions. Proper exception handling is essential to ensure that the application behaves in a predictable and reliable manner, even when errors occur.

Steps:

1. Identify the codebase and all modules in the Wagtail project that handle user input, perform complex operations, or interact with external systems.
2. Review the codebase to identify any instances where exceptions are thrown or error conditions are handled.
3. Verify that the exceptions or error conditions being handled are appropriate for the context in which they occur, and that they account for both expected and unexpected error conditions.
4. Test the application by deliberately causing errors or unexpected conditions, using appropriate test data and tools.

5. Verify that the application handles these errors and conditions appropriately, such as by logging error messages or providing users with informative error messages.
6. Verify that the application behaves in a predictable and reliable manner even when errors occur.
7. Verify that the application meets industry best practices for exception handling, such as logging errors and preventing information leakage.
8. Document any issues found during testing and report them to the development team for follow-up testing.

Expected Result: The expected result of this test is that the Wagtail project's codebase uses consistent and appropriate exception handling throughout, accounting for both expected and unexpected error conditions. The application should handle errors and unexpected conditions gracefully, logging error messages where appropriate and providing users with informative error messages. The application should behave in a predictable and reliable manner even when errors occur and should meet industry best practices for exception handling.

Actual Result: A lot of releases are being made to handle errors properly. For example the search implemented in the backend is Elasticsearch which was unable to roll back if elasticsearch version <5.4.x was used as Wagtail deletes the index which was present before the search and rebuild it completely.

To mitigate this issue, the Elasticsearch version is upgraded to 5.5.x which has a setting called ATOMIC_REBUILD which rebuild into a separate index while keeping the old index active until the new one is fully built.

7. Test Case ID: ASVS V14.4.5 – HTTP Security Headers

CWE: 523 - Verify that a Strict-Transport-Security header is included on all responses and for all subdomains, such as Strict-Transport-Security: max-age=15724800; includeSubdomains.

Description: The HTTP Strict Transport Security (HSTS) feature lets a web application inform the browser through the use of a special response header that it should never establish a connection to the specified domain servers using un-encrypted HTTP. Instead, it should automatically establish all connection requests to access the site through HTTPS. It also prevents users from overriding certificate errors.

Steps:

1. Run the wagtail project on localhost, Port 8000
2. In Git Bash, type in this command: `curl -s -D- http://localhost:8000 | grep -i strict`
3. Check output

Expected Result: After typing in the curl command for the website, an output was supposed to be generated:

```
rvss2@Ronin MINGW64 /e/NCSU/Software_Security/labcorp
$ curl -s -D- https://owasp.org | grep -i strict
strict-transport-security: max-age=31536000; includeSubDomains
```

(Here we are testing on the owasp website.)

Actual Result:

```
rvss2@Ronin MINGW64 ~
$ curl -s -D- http://localhost:8000 | grep -i strict
rvss2@Ronin MINGW64 ~
```

No output was generated, meaning the website does not have the HSTS header implementation.

8. Test Case ID: ASVS V12.1.1 – File upload

CWE: 400 - Verify that the application will not accept large files that could fill up storage or cause a denial of service.

Description: The purpose of this test is to ensure that the application does not accept files consuming huge amounts of memory, which can lead to denial of service. Here, we have taken into consideration a Zip bomb – which is an archive file containing huge amount of data. Its intent is to cause a denial of service by exhausting the disk space or memory of the target system that tries to extract the archive.

Steps:

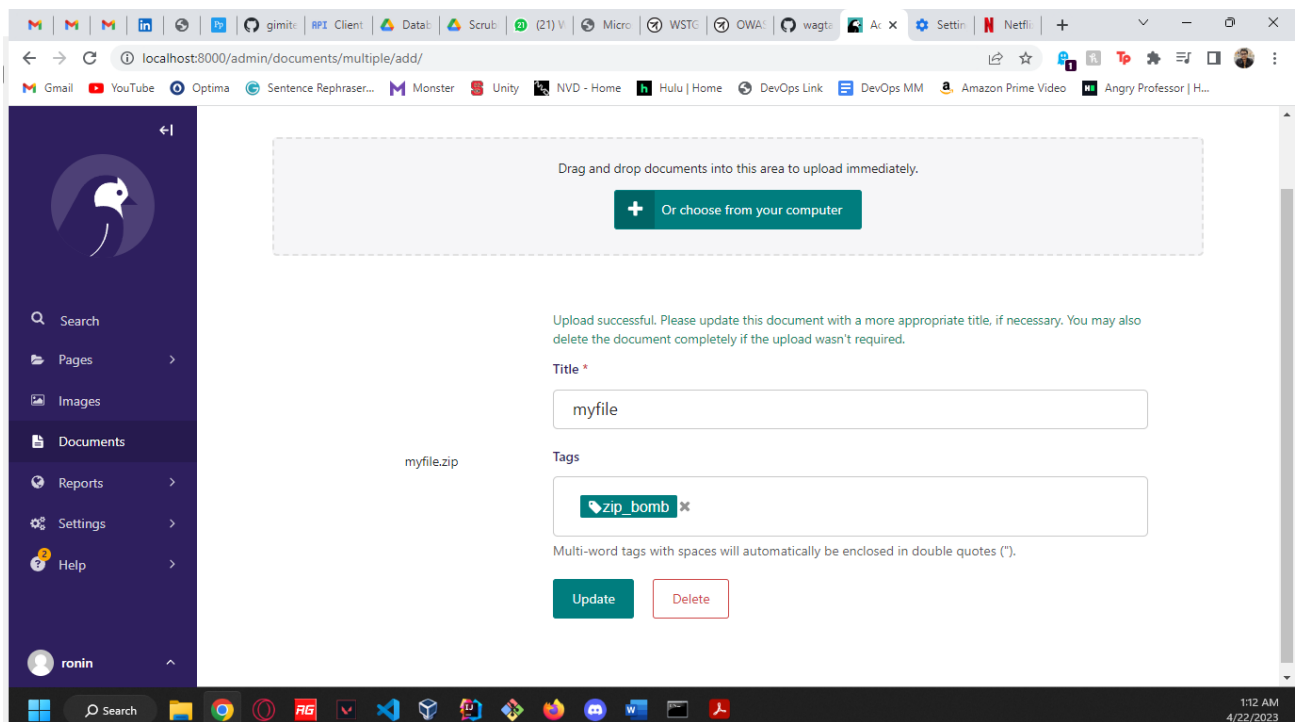
1. Create a large file consisting of a single character. In this demonstration we create a file that will be decompressed to 1GB. This command below is performed in Git Bash

```
rvss2@Ronin MINGW64 /e/NCSU/Software_Security/labcorp
$ dd if=/dev/zero of=myfile.bin bs=1000M count=1024
dd: error writing 'myfile.bin': No space left on device
242+0 records in
241+0 records out
252706816000 bytes (253 GB, 235 GiB) copied, 1790.97 s, 141 MB/s

rvss2@Ronin MINGW64 /e/NCSU/Software_Security/labcorp
$ zip myfile.zip myfile.bin
updating: myfile.bin (172 bytes security) (deflated 100%)

rvss2@Ronin MINGW64 /e/NCSU/Software_Security/labcorp
$ _
```

2. Upload the file by selecting upload from computer in the Documents section in the website.



Expected Result: The system should pose an error message due to the file size limit being exceeded, and not proceed with the upload.

Actual Result: The file was uploaded, posing a threat for Denial of Service.

