DAT250 - Report

PART 1

Liv Underhaug	238013
Mina Wøien	256461
Åse J. Sagebakken	254340

October 23, 2020

Contents

Introduction

The objective of this project have been to develop and secure a website. We have used python flask for the backend, HTML for flask templates and SQLAlchemy for storing data. To create the general website we have have applied a tutorial. To secure it, we have used OWASP top 10 as a framework. To collaborate on the code, we have used GitHub repository. Link for GitHub: `https://github.com/livun/photomind_2`

The application we chose to create was a blog, called Blogmind. There you can register a user, update your account, create, update and delete posts and reset your To host the website we have used Heroku, and the link to the side is `https://blogmind.herokuapp.com/`

Sitemap & Threat Model

We have included a sitemap and a threat model. The site map provides information about the pages and the relationships between them. The treat model is a process that reviews the security of our system, we have identified the data flow and the trust boundaries.

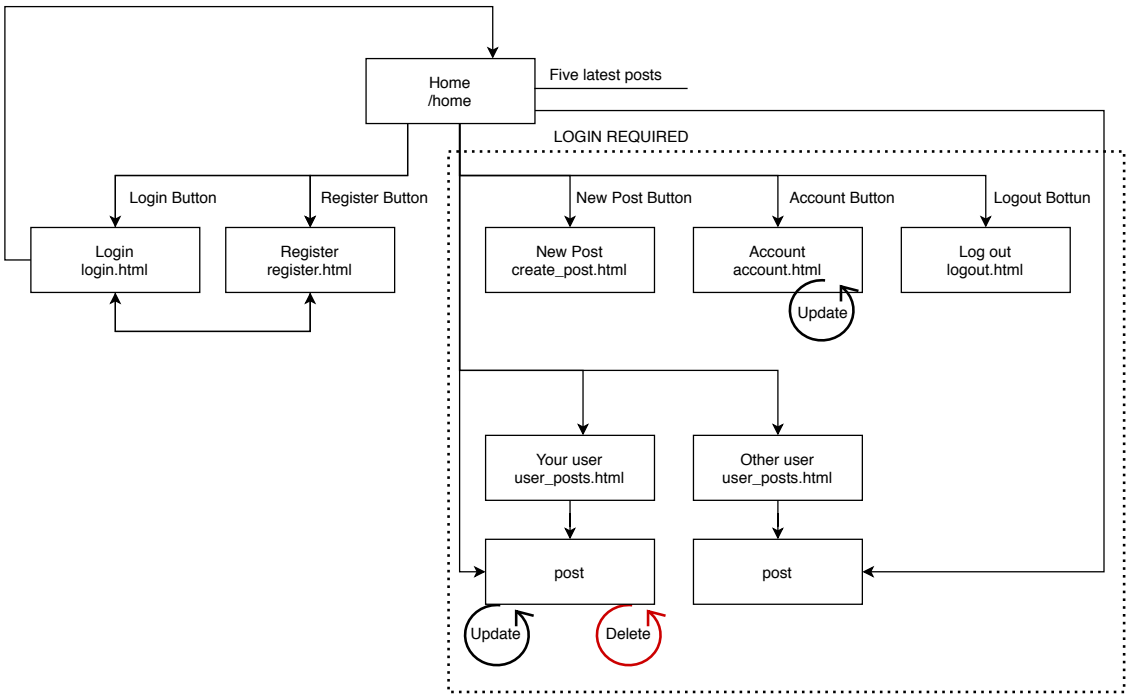

Figure 1: Sitemap

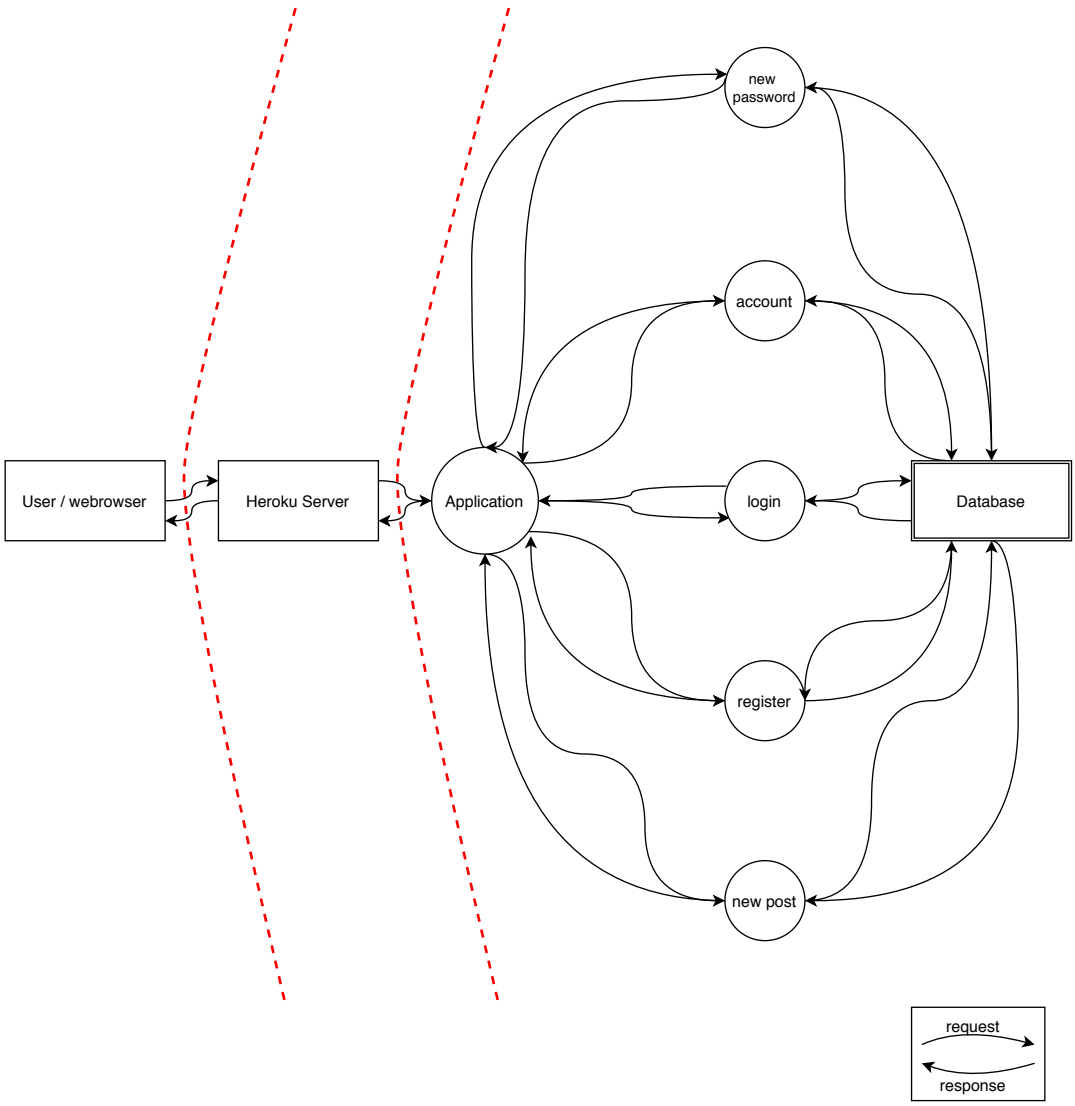

Figure 2: Threat Model

OWASP TOP 10

1. Injection

"Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query [9]. The site is vulnerable to injection if there are possible input fields, where an attacker can submit code and and trick the software to run that input.

As we use a database to store all data on our site, the possibility for a SQL-injection is present. This is a vulnerability that allows an attacker to change SQL queries made to the database, and therefore retrieve sensitive information.

SQL Injection

We have used SQLALchemy for our database, as it is claimed to be SQL injection proof. However, we have chosen to do some additional actions to prevent injection attacks. First, we have made sure that we haven't written any SQL query's in plain-text. We have only used prepared statements in classes and functions, see code 1 & 2 for examples.

Code 1: users/models.py

```
9  @login_manager.user_loader
10 def load_user(user_id):
11     return User.query.get(int(user_id))
```

Code 2: users/routes.py

```
9  def login():
10     ...
11     form = LoginForm()
12     if form.validate_on_submit():
13         user = User.query.filter_by(email=form.email.data).first()
```

Second, by implementing the escape() function, as shown in code 3, in all input fields at the site, it converts special characters into HTML-safe sequences. The escape function don't catch all special characters. For the remaining, we have blacklisted them, see code 4 and 5. With these two measures we have made it difficult to inject query's that bypasses authentication. According to OWASP, whitelisting is the better way to go, but in our case we chose blacklisting for demonstrating the effect of it.

Code 3: users/routes.py

```
25 def register():
26     ...
27     user = User(username=escape(form.username.data), ...
           email=form.email.data, password=hashed_password, ...
           question=form.question.data, answer=hashed_answer)
28     ...
```

Code 4: forms.py

```
8  avoid = ['#', '$', '%', '/', '(', ')', '=', '{', '}', '[', ']', ':', ';', ...
       '\\', '*', '^', '  ', '~', '’']
```

Code 5: users/forms.py

```
25 class RegistrationForm(FlaskForm):
26     ...
27     def validate_username(self, username):
28         ...
29         for i in avoid:
30             if i in username.data:
31                 raise ValidationError('Special characters is not allowed, ...
                       try again.')
```

2. Broken Authentication

Authentication is the process of verifying that an individual, entity or website is who they claim to be. Broken authentication attacks happens when a hacker is able to overtake user accounts, so he gets the same privileges as the user. This could be a risk when functions related to authentication are not implemented correctly, allowing attackers to get a hold of passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.

Knowledge Based Authentication

To prevent hijacking and protecting the users profile, there is implemented Knowledge Based Authentication (KBA). The KBA has two part, a password and a security question. The password is used for login and the security question is designed for a password reset. In the registration process, the user will be asked to provide a password with the minimum length of 16 characters. A long password makes it harder to crack and therefore also more secure. Even though we encourage the users to have long passwords, we have set the maximum length of password to 64 characters to prevent cross site scripting, as shown in code 6.

Code 6: users/forms.py

```
9  class RegistrationForm(FlaskForm):
10     ...
11     password = PasswordField('Password', validators=[DataRequired(), ...
           Length(min=16, max=64)])
12     ...
```

Next, the user will be asked to choose a question from a list of five unique security questions, and provide a unique answer. The questions are written in such a manner that the possibility for duplicates and standard answers is rather low.

- What was the house number and street name you lived in as a child?
- What is your oldest cousin's first name?
- What primary school did you attend?
- In what town or city was your first full time job?
- What is the middle name of your oldest child?

The password reset function is accessible before login, and will ask for an email and an answer to a chosen security-question and a new password. The email, security question and answer have to match with the original set in the registration process, for the password to be reset. Both password and answer are salted and hashed before stored in the database, in case an attacker accesses it.

This is a simple way to add security and protect the users profile. The next step would be to implement a two-factor authentication or a multi-factor authentication.

Multi Factor Authentication

Multifactor authentication (MFA), or Two-Factor Authentication (2FA) is when a user is required to present more than one type of evidence in order to authenticate on a system. There are four

different types of evidence that can be used, listed below:

- Something you know (as passwords, security questions)
- Something you have (as hardware or software tokens, email, SMS)
- Something you are (as fingerprints, facial recognition)
- Location (as source IP ranges, geolocations)

According to Symantec's 2016 Internet Security Threat Report, 80% of breaches can be prevented by using multi-factor authentication. Thus, by using basic, two-factor authentication, an organization can immediately reduce its cybersecurity threat profile in a fast and meaningful way.[10]

Automatically Logout

An other measure to protect users' sessions from being hijacked and their profile being abused, is to implement an automatic inactivity logout. If the user have been inactive for 20 minutes, the user will be logged out and the session will be deleted. To change the permanent session lifetime, we configure the session cookie, as shown in code 7, and set the time to 20 minutes. In code 8, we set the session permanent to True for the login function.

Code 7: config.py

```
2 class Config:
3     ...
4     PERMANENT_SESSION_LIFETIME =  timedelta(minutes=20)
```

Code 8: users/routes.py

```
2 def login():
3      ...
4      session.permanent = TRUE
5      ...
```

Limited Failed Login Attempts

An other measure to prevent attackers from brute forcing into to users' accounts, is to use a rate limiter on the various routes. The key is the attacker or users IP address, and the limiter restrict possible attempts on the login route, the user register route and the new-password route.

Code 9: init.py

```
6  from flask_limiter import Limiter
7  from flask_limiter.util import get_remote_address
8
9  limiter = Limiter(key_func=get_remote_address)
10
11 def create_app(config_class=Config):
12     ...
13     limiter.init_app(app)
```

The code below shows how many attempts that are possible, before the IP gets locked out from the route. If the number of failed login attempts corresponds to the limiter, the IP address will be locked out. If the user have forgotten the password, it can be reset, but only one time a day. This prevents exploiting of the new-password route as well.

Code 10: users/routes.py

```
10 @users.route("/register", methods=['GET', 'POST'])
11 @limiter.limit('5/day')
12 def register():
13     ...
14
15 @users.route("/login", methods=['GET', 'POST'])
16 @limiter.limit('50/day; 20/hour; 5/minute')
17 def login():
18     ...
19
20 @users.route("/newpassword", methods=['GET', 'POST'])
21 @limiter.limit('1/day')
22 def newpassword():
23     ...
```

3. Sensitive Data Exposure

This vulnerability appears from many different attacks. The most common attack is "Man in the middle attack", where an attacker is able to place himself between a sender and a receiver of data, and read or change it. Therefore, we do not store any data that we don't need, as it is the best way to protect it. Sensitive data can't be stolen if we don't have it in the application. However, some data needs to be stored, so we have implemented measurements to provide sensitive data exposure.

Encrypting

For protection against the exposure of sensitive data to someone who shouldn't have access to it, we encrypt the data before storing it in the database, shown in code 11. When a user registers he creates a password and an answer to a chosen question, which is hashed before stored. However hashing have been easy to crack with modern graphics processing units, therefore we use bcrypt, which is design to de-optimize such vulnerabilities. For sensitive data that must be protected, as password and answer in our application, bcrypt is recommended.

The bycrypt automatically generates a salt and appends it to the hashed password to create an unique hash for every input. To generate randomness in the hashing, the salt is randomly generated sting added to the password. Even though the passwords have the same string, they will have different output values from bcrypt.

Code 11: users/routes.py

```
30 def register():
31     ...
32     hashed_password = ...
           bcrypt.generate_password_hash(escape(form.password.data)).decode('utf-8')
33     hashed_answer = ...
           bcrypt.generate_password_hash(escape(form.answer.data)).decode('utf-8')
34     ...
```

If an attacker should have been able to access our database, he would only be able to see the hashed passwords and answers, which is the only sensitive data stored. There isn't any unnecessary stored data, as it is either needed for a secure login or for the website's functions.

Autocomplete and caching

To prevent an attacker from finding an email, username or answer to one of the questions, the autocomplete is disabled on the login, register and create new password forms. This is done by setting the autocomplete off in the html-forms, as shown in code 12.

Code 12: templates/login.html

```
4 ..
5 <form method="POST" action="" autocomplete="off">
6 ...
```

To disable caching on forms that collect sensitive data, we have made sure that it requires login for certain sites. If a user logs out, you can't press the back button without logging in again. This is done by implement the nocache() function as shown in code 13.

Code 13: nocache.py

```
5  def nocache(view):
6      @wraps(view)
7      def no_cache(*args, **kwargs):
8          response = make_response(view(*args, **kwargs))
9          response.headers['Last-Modified'] = datetime.now()
10         response.headers['Cache-Control'] = 'no-store, no-cache, ...
               must-revalidate, post-check=0, pre-check=0, max-age=0'
11         response.headers['Pragma'] = 'no-cache'
12         response.headers['Expires'] = '-1'
13         return response
14
15     return update_wrapper(no_cache, view)
```

4. XML External Entities (XXE)

This attack happens when a weakly configured XML parser processes an XML input containing a reference to an external entity. As the website doesn't let the users upload an XML file, preventing this type of attack isn't relevant for our application.

5. Broken Access Control

Broken access control occur when an attacker or user gets access to functions and data on the application that they are not authorized to. The implication of this is that attackers can act as users or administrators, or users using privileged functions, or creating, accessing, updating or deleting every record [9].

Role-Based Access Control (RBAC)

To enforce access control on the site, we have implemented role based access control. The roles are set in the database, and are either 'admin' or 'user'. The default when signing up is user and he can create, edit and delete his own posts, and only view others' posts. The admin role is set manually and code 14 shows how access is given to a separate admin page with all privileges.

Code 14: models.py

```
5  class MyModelView(ModelView):
6      def is_accessible(self):
7              return current_user.role == 'admin'
8
9      def inaccessible_callback(self, name, **kwargs):
10         return redirect(url_for('users.login'))
```

To prevent attackers trying bypass access control by modifying the URL, we have used flask-login's function *is_authenticated()*, and decorators *@login_required* and *@fresh_login_required*.

6. Security Misconfiguration

To prevent security misconfiguration such as clickjacking, we are using OWASP secure headers project. This describes the HTTP response headers which makes our application more secure, as the HTTP response headers restrict modern browsers from easy vulnerabilities. To implement this, we have made a function who runs after every route request. See code 15.

Code 15: routes.py

```
14 @main.after_request
15 def headers(response):
16     response.headers['Strict-Transport-Security'] = 'max-age=31536000; ...
           includeSubDomains'
17     response.headers['Content-Security-Policy'] = "default-src 'self' ...
           https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
18     response.headers['X-Content-Type-Options'] = 'nosniff'
19     response.headers['X-Frame-Options'] = 'DENY'
20     response.headers['X-XSS-Protection'] = '0'
21     response.headers['X-Permitted-Cross-Domain-Policies']='none'
22     response.headers['Referrer-Policy']='strict-origin-when-cross-origin'
23     return response
```

HTTP Strict Transport Security (HSTS)

To protect our website against cookie hijacking and typically man-in-the-middle attack, we have used the HTTP strict transport security. It makes web servers make the web browsers to only interact with the secure HTTPS and not the unsecure HTTP. To block the access to sites that only can serve with HTTP, we set all the present and future subdomains to a year, see code 15.

X-Frame-Options

The X-Frame-Options helps provide the website against clickjacking, as it decides whether or not a side should be allowed to render an object or frame. This header is set to deny, as shown in code 15, and it ensures that the content isn't embedded into other sides.

X-Content-Type-Options

To provide against MIME sniffing vulnerabilities, as upload a file that can help them perform cross-site-scripting, we can set the X-Content-Type-Options to nosniff, se code 15. This will not have any effect if the user uses older versions of Chrome or Internet Explorer, as these browsers doesn't support the X-Content-Type-Options.

Content-Security-Policy

The "default-src: 'self'" is intended to prevent cross-site-scripting. We want all content to come from the website's origin, except the css-bootstrap and the profile pictures.

X-Permitted-Cross-Domain-Polices

To prevent integration from Flash and PDF documents by different domains, we have set the X-Permitted-Cross-Domain-Policies and set it to none, as shown in code 15.

Referrer Policy

To protect the users sensitive information from input fields from current site to the redirection site, we set the Referer Policiy to strict-origin-when-cross-origin, as shown in code 15. When preformin a same-orgin request, it sends the origin, path, and querystring. However the orgin is only sendt when the protocol security level stays the same while performing a cross-origin request. [15]

X-XSS-Protection

The X-XSS-Protection isn't used by modern browsers because this can cause several security issues. Therefore OWASP has recommend to set it to '0', see code 15.

7. Cross-Site Scripting XSS

When an application includes untrusted data and allows attacker to execute scripts in the users' browser. To prevent users to get redirected to unintended destinations or session hijacks, we have implemented the function escape() and a blacklist, as explained in chapter 1. An attacker won't be able to write a malicious script in the post, as the characters will be replaced by text, see code 4 and 16.

Code 16: posts/routes.py

```
25 def new_post():
26     ...
27     post = Post(title=escape(form.title.data), ...
           content=escape(form.content.data), author=current_user)
28     ...
```

By using the FlaskForm framework we make the cross-site request forgery vulnerabilities easier to avoid, as the form already is getting the CSRF protection.

Another way we are preventing XSS is by not allowing any users to upload any files, except the profile picture which is set to only allowing "jpg", "jpeg" and "png". This means that an attacker won't be able to upload a file with malicious script.

8. Insecure Deserialization

Insure Deserialization is an attack where an manipulated object is injected into the context of the application. This attack may result in SQL injection or high-severity attacks. Since we don't use serialization, preventing an attack like this isn't relevant for our website.

9. Using Components with Known Vulnerabilities

Two prevent this type of vulnerability, all the installed packages we use in the application have the most recently available version. By using the CVE we are able to check all the packages we use in the application, for known vulnerabilities.

By using the latest version of the packages, we can avoid injection. Earlier versions of SQLALchemy allowed SQL injection through the order_by parameter and when the group_by parameter can be controlled.

Types of cross-site scripting can also easily be avoided, by using a newer version than 0.11.11 of Werkzeug. As the render_full function in debug/tools.py allowed attackers to inject web script or HTML through a field that contained an exception message. The version of Mako before 0.3.4 also had a cross-site vulnerability, as the package relied only on the escape function. This made it easier for attackers to conduct XSS attacks.

10. Insufficient Logging & Monitoring

This is not the cause of why you get hacked, however, vulnerabilities of insufficient logging and monitoring happens when security-critical applications aren't logged properly, and the system doesn't monitor this properly. If the application were to be hacked, without, we wouldn't be able to see what went wrong.

To prevent this vulnerability, we have implemented flask_admin. Which is an admin interface, were we are monitoring all of our models. With this action we can see who is active, last login, and IP, see code 17

Code 17: models.py

```
15 class User(db.Model, UserMixin):
16     ...
17     last_login_at = db.Column(db.DateTime)
18     last_login_ip = db.Column(db.String(100))
19     active = db.Column(db.Boolean)
20     ...
```

To extend this interface and improve on our monitoring the next step would be to log and get notified when an attacker tries to brute force into site.

Picture Handling

Even though malicious files aren't one of the OWASP top 10, we have chosen to consider the possibility for attacks by file uploads.

To prevent an attacker from uploading a malicious file when uploading a profile picture, the only allowed file types are set to "jpg", "jpeg" and "png". The profile picture also has an image size maximum to prevent the website from denial of service attack, and in order to protect the file storage capacity, see code 18.

Code 18: config.py

```
25 class Config:
26     ...
27     ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg'}
28     MAX_CONTENT_LENGTH = 1024 * 1024  # 1MB max-limit
29     ...
```

Because of the principle called *never trust user input*, it is important to secure the filename of a uploaded file before storing it. All submitted form data can be forged, and filenames can be dangerous. The function *secure_filename()* is used to sanitizes the filename before storing it. [5]

Code 19: users/routes.py

```
25 filename = secure_filename(file.filename)
```

Conclusion

The project have been instructive. We have been challenged by developing a website for the first time, as we didn't have any experience with web development and data storing. We have learned how to use flask to develop a website, and use several flask packages to help secure it. If we had had more time we would have continued to find vulnerability's to prevent and secure. However, we have implemented as many OWASP security holes as the time allowed us to.

Bibliography

[1] Bcrypt: Choosing a work factor. https://flask-bcrypt.readthedocs.io/en/latest/.

[2] Flask talisman. https://pypi.org/project/flask-talisman-rdil/.

[3] Owasp top 10 - 2017 the ten most critical web application security risks. https://owasp.org/www-project-top-ten/.

[4] Referer header: Response header hardening. https://www.scip.ch/en/?labs.20180308.

[5] Uploading files. https://flask.palletsprojects.com/en/1.1.x/patterns/fileuploads/, 2010.

[6] Bcrypt: Choosing a work factor. https://wildlyinaccurate.com/bcrypt-choosing-a-work-factor/, 2012.

[7] Disabling caching in flask. https://arusahni.net/blog/2014/03/flask-nocache.html, 2013.

[8] Flask series: Security. https://damyanon.net/post/flask-series-security/, 2015.

[9] Owasp top 10 - 2017 the ten most critical web application security risks. https://raw.githubusercontent.com/OWASP/Top10/master/2017/OWASP%20Top%2010-2017%20(en).pdf, 2017.

[10] Broken authentication attack. https://www.immuniweb.com/blog/OWASP-broken-authentication-attack.html, 2018.

[11] Content security policy (csp). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP/, 2020.

[12] Owasp secure headers project. https://owasp.org/www-project-secure-headers/, 2020.

[13] Owasp secure headers project. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=flask, 2020.

[14] Referer header: privacy and security concerns. https://developer.mozilla.org/en-US/docs/Web/Security/Referer_header:_privacy_and_security_concerns, 2020.

[15] Referer policy. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy, 2020.

[16] Thomas Varghese. Addressing red flags compliance. https://www.scmagazine.com/home/opinions/addressing-red-flags-compliance/, January 2009.