Broken Authentication

White Hats: Team Members
Banks Cargill - cargillb@oregonstate.edu
Frederick Eley - eleyf@oregonstate.edu
Timothy Glew - glewt@oregonstate.edu

Description

The second web application security risk listed by the OWASP Foundation is broken authentication. Broken authentication occurs when an attacker either steals login data or forges session data, such as cookies, in order to gain access to a webapp. Once an attacker has broken into a user's account they can view secure information such as credit card numbers, addresses, birthdates, social security numbers, etc.. With this information, an attacker could then perform further nefarious actions such as social security fraud, identity theft, or simply disclosing highly sensitive information to others (owasp.org).

There are a number of vulnerabilities that need to be strengthened to protect against broken authentication. The degree that each web app strengthens itself is dependent on the information being stored. Session management is the most basic protection. Session management refers to the limitation of session duration from either login or the last action taken. It ensures that if a shared computer is utilized and the user neglects to logout, the next user has a reduced chance of being able to access the account (sitelock.com).

Another part of session management is ensuring that session hijacking is protected against. Session hijacking occurs when a hacker monitors your network activity while you're engaged in an active session. If they are able to parse the TCP packets, they have the potential to record your session information and reuse it, or to potentially interrupt and take control of your session. The most basic defense is ensuring that you never access sensitive data when using an exposed network. If you need to access it, then use a VPN to protect yourself.

The next major vulnerability that needs to be handled is protection against credential stuffing. Credential stuffing is where the attacker uses a script to brute force or automate an attack using a list of valid usernames and passwords. The most common defense against this is ensuring that your users choose passwords that are strong where strength of a password is primarily dictated by its length followed by the number/type of characters(password-depot.de).

Password recovery is the last vulnerability that the majority of web applications should be strengthening. Security based questions used to be commonplace to recover login credentials but they have been proven to be impossible to defend based on research from two security researchers. They state that "secret questions are neither secure nor reliable enough to be used as a standalone account recovery mechanism." They argue this based on an underlying flaw of secret questions; they are either memorable or "somewhat secure", but almost never both (security.googleblog.com).

The strongest form of protection against broken authentication is multifactor authentication, which is an authentication method that requires a user to present at least two pieces of

evidence that they are the user they claim to be. The most common is a 2FA or two-factor authentication which typically combines: 1) something the user knows, 2) something the user has. A common example for this is an ATM transaction. The user must have and enter the card as well as know the PIN associated with the card to be granted access. While multifactor authentication provides the most security against broken authentication, it is also inconvenient to the user. Therefore it hasn't yet become common practice unless the information stored has been deemed as excessively high-risk.

How to Attack our Site

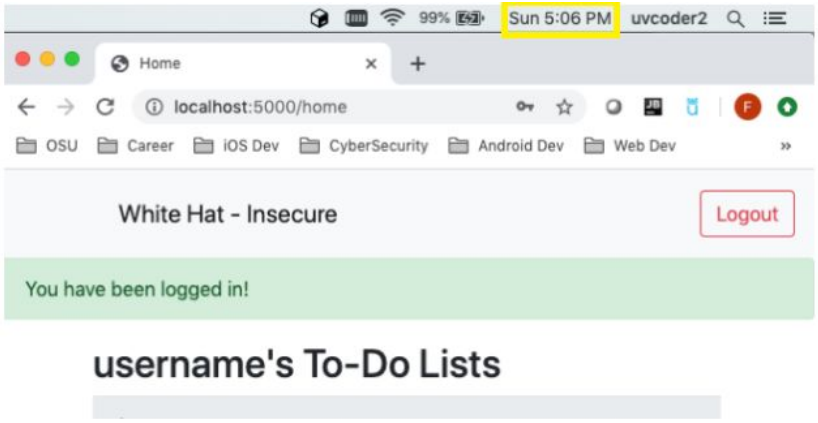
Session Management

Session ID Inactivity

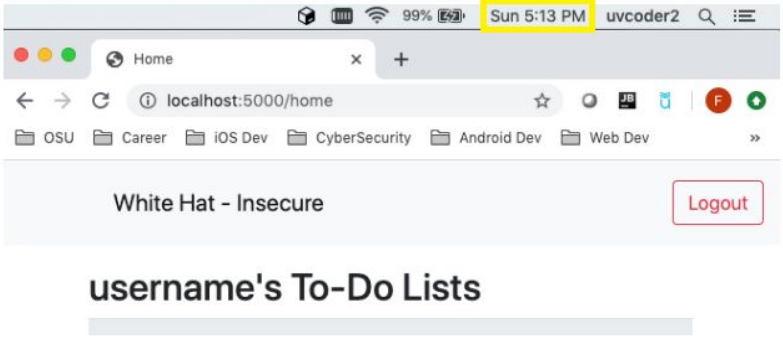
Our insecure web application does not log a user out after a period of inactivity. A user may presume that because a window has been closed that their session has been logged out. Without explicitly logging out someone else may use the same computer and log into the previous users account without needing to log in.

To see this flaw login to your created account or a default account. Without logging out close the browser window and wait 5 minutes. Upon reopening the web page you will be directed to the home page of the last logged in user.

User's initial log in: 5:06PM



After closing the browser the same user remains logged in more than five minutes later: 5:13PM



Passwords

Uncomplexed passwords

Our insecure web application does not enforce any rules for password complexity. Previous breaches of web sites have produced lists of commonly used passwords such as “password”, “qwerty”, and “abc123”. There are also lists of default and common usernames. An attacker can easily utilize automation methods to try thousands of these username and password combinations to try to guess their way into an account.

Here you can see our method of acquiring the password on the insecure site to register a new user. There is no requirement implemented for password complexity meaning a user could choose a single character as their password, which would be incredibly easy for a script to hack.

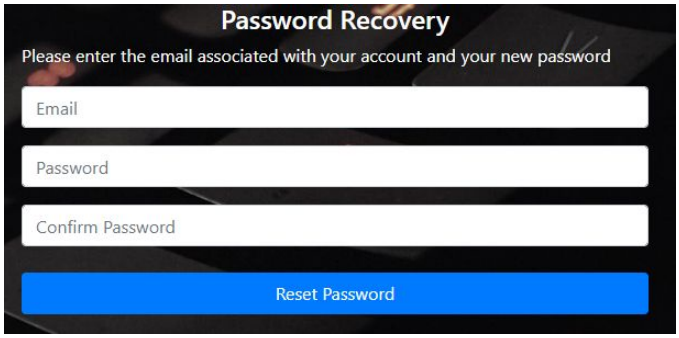
```
if request.method == 'POST':

    email = request.form['email']
    username = request.form['username']
    password = request.form['password']
    confirm_password = request.form['confirm_password']

    if password != confirm_password:
        flash('Password confirmation does not match password', 'danger')
        return render_template('accountCreation.html')
```

To visualize this flaw, from the login page click the link to “Sign Up”. On the “Sign Up” page create an account and use the simple password “123456”. Once back on the login page, sign into the newly created account with the weak password.

Recovery

To recover your password on our insecure site, the only requirement is that you must know an email address that is in the database. Using SQL injection, an attacker could easily return a single row and change its user’s password (read our how-to for SQL injection for more information). Since there are no additional securities to prevent an attacker from testing multiple different emails in rapid succession, credential stuffing would be incredibly effective here as well. For more on credential stuffing, continue below.

Only the email is required in order to reset the password:




```
# make sure email is unique
query = 'SELECT `email` FROM users'
cursor = execute_query(db_connection, query)
rtn = cursor.fetchall()
cursor.close()
if (not any(email in i for i in rtn)):
    flash('Email not registered, please try again', 'danger')
    db_connection.close() # close connection before returning
    return render_template('passwordRecovery.html')
```

```
query = ('UPDATE `users` '
         'SET pword = %s WHERE email = %s;')
data = (password, email)
```

Passwords in plaintext (sensitive data)

Our insecure web application stores passwords in the database in clear text. If an attacker were able to use a SQL injection attack (see SQL Injection write-up) and dump a table they would have full access to every users credentials. The only confidential information stored by our app is the users password but the same would apply to any other personal confidential information such as credit card and social security numbers.

2 rows | Tx: Auto | Tab-se...d (TSV)

<Filter criteria>

	user_id	username	pword	email
1	1	username	password	username@password.com
2	2	Anonymous	123456	Anonymous@123456.com

Credential Stuffing

Brute Force

To perform brute force credential stuffing on our insecure site we can use a tool called Hatch. Hatch automates the process of attempting to log in to a website using a specified username and a list of possible passwords. An extended tutorial of how to use Hatch can be found here:

https://null-byte.wonderhowto.com/how-to/brute-force-nearly-any-website-login-with-hatch-0192225/

To use Hatch:

1. Make sure you have Python 2 installed and then install: 'selenium' and 'requests'
    a. '$ pip2 install selenium'
    b. '$ pip2 install requests'
2. Download ChromeDriver and put the downloaded file into a directory named '**webdrivers**' on your C drive
    a. http://chromedriver.chromium.org/downloads
3. Clone the Hatch repo
    a. https://github.com/nsgodshall/Hatch.git
4. Change to the directory where you cloned Hatch and run:
    a. '$ python2 main.py'
5. You will be prompted to enter the following:
    a. $ Enter a website: **https://osu-capstone-project-insecure.herokuapp.com/**
    b. $ Enter username selector*:* **body > div.container > div.row.mt-5 > div.col-8 > div > form > div:nth-child(1) > input**
    c. $ Enter password selector: **body > div.container > div.row.mt-5 > div.col-8 > div > form > div:nth-child(2) > input**
    d. $ Enter the login button selector: **body > div.container > div.row.mt-5 > div.col-8 > div > form > div:nth-child(3) > button**
    e. $ Enter the username to brute-force: **username**
        i. *'username' can be replaced with the username for which you would like to brute force, but for this example 'username' is a registered user with an insecure password that is included in 'passlist.txt' discussed in step 5f.*
    f. Enter a password list: **passlist.txt**
        i. *'passlist.txt' is a list of commonly used passwords that comes with the Hatch repo cloned in step 3. This file can be replaced by your own list of passwords or otherwise modified to suit your needs.*
6. Hatch will then attempt to log in to the username specified in step 5e sequentially with the list of passwords specified in step 5f until every password in the list is tried, the password is guessed, or you have been locked out of attempting to log in.
7. If the password for the specified username was included in the password list, once Hatch finishes running you will be logged into the site and the correct password will be displayed in the terminal.

How to Defend our Site

Session Management

ID Timeout

To defend against ID timeout, we implemented a permanent session lifetime for all logged in users of 10 minutes. Each time they take an action on the web application their session is modified with an updated time, allowing the session to be current for another 10 minutes.

Global variable for the webapp:

```
# sets the session timeout to 10 minutes
webapp.permanent_session_lifetime = timedelta(minutes=10)
```

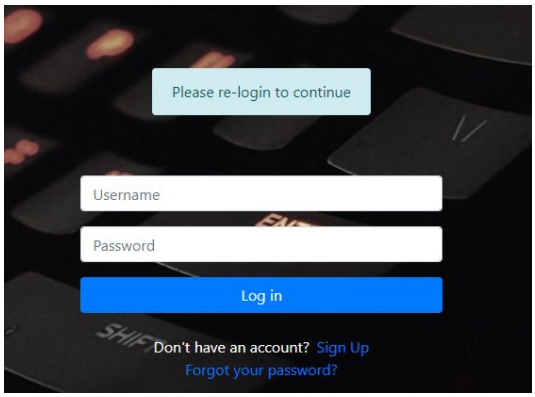
Function that is called before each request is made updating their session.

```
@webapp.before_request
def before_request():
    session.modified = True
```

Set the user's session to permanent after they have successfully logged in.

```
if username == result[0][1] and password == result[0][2]:
    user = User(user_id=result[0][0], username=result[0][1], password=result[0][2], email=result[0][3])
    login_user(user)
    session.permanent = True
    flash('You have been logged in!', 'success')
    next_page = request.args.get('next')
    db_connection.close() # close connection before returning
    return redirect(url_for('home'))
```

If a user's session does timeout and they are required to re-login, they are returned to the login page using a flask_login module for LoginManager. It allows us to use the decorator, @login_required, before each function which requires the user be logged in. When they are returned to the login page, they are flashed a message giving them feedback that they must re-log in.



Login_manager allows us to set the page we want rendered, the message to be flashed and the type of message, which set the color of the flashed message.

```
login_manager.login_view = '/login'
# message and cateogry that are flashed when session expires
login_manager.login_message = "Please re-login to continue"
login_manager.login_message_category = "info"
```

Example: decorator before function that navigates to the home page

```
@webapp.route('/home')
@login_required
def home():
    """
    Route for the home page of a user where all of their to-do lists will be listed
```

Passwords

Strengthening

In order to defend against weak password credentials we created a function to verify that a user's password contains at least 8 characters with at least one lower case letter, uppercase letter, number, and special character. This combination leads to 95^8 or 6.6342043e+15 possible password combinations.

```
def complex_password(password):

    if len(password) >= 8 and \
            any(char.isdigit() for char in password) and \
            any(char.islower() for char in password) and \
            any(char.isupper() for char in password) and \
            any(char.islower() for char in password) and \
            any(not char.isalnum() for char in password):
        return True
    else:
        return False
```

This function is implemented in the registration page upon submission.

```
email = request.form['email']
username = request.form['username']
password = request.form['password']
confirm_password = request.form['confirm_password']

if not complex_password(password):
    flash('Password requirements not met', 'danger')
    return render_template('accountCreation.html')

if password != confirm_password:
    flash('Password confirmation does not match password', 'danger')
    return render_template('accountCreation.html')
```

Recovery
Password recovery defense

Credential Stuffing

Brute Force

To strengthen our secure site against brute force attacks attempting to log in to our site, if an account fails to provide the correct password three times in five minutes they will be locked out of the account for 5 minutes. With this restriction in place brute force attacks will be drastically less effective since the attacker will be limited to three guesses per five minutes. This makes it much more difficult for a brute force attack to succeed since it will take significantly longer to try possible passwords.

One potential drawback of implementing an account lockout is that a user who is the target of a brute force attack could have their account locked at no fault of their own; however, the lockout is only 5 minutes, so we believe this risk and inconvenience is worth the added security.

To implement this we modified our login route in the following ways:

Query for timestamp of the last login attempt for the username and calculate the elapsed time between that timestamp and the current time.

```
# if the user provided a valid username
if result:
    # get information about login attempts
    last_login_attempt = result[0][7]  # get last login attempt datetime
    current_time = datetime.now()  # get current datetime
    difference = current_time - last_login_attempt  # calculate the difference
    seconds_in_day = 24 * 60 * 60
    # convert difference to a tuple of difference in minutes and seconds
    difference = divmod(difference.days * seconds_in_day + difference.seconds, 60)
```

If there have been more than 3 failed attempts and less than 5 minutes have elapsed since the third failed attempt, do not process the login information and display an error.

```
# if they've failed more than 3 attempts in the last 5 minutes, don't allow login
if result[0][6] >= 3 and difference[0] < 5:
    flash('Too many failed login attempts. Try again later', 'danger')
    db_connection.close() # close connection before returning
    return render_template('login.html')
```

Else, if the provided username and password match what is stored for the user, then the login is successful, so their ‘login_attempts’ is reset to 0 and their ‘last_login_attempt’ is updated before logging them in and redirecting to the home page.

```
# else check validation that user input matched query results - successful login
elif username == result[0][1] and password == result[0][2]:
    # reset login_attempts to 0
    query = "UPDATE users SET login_attempts = 0 WHERE user_id = '{}'".format(result[0][0])
    cursor = execute_query(db_connection, query)  # run query
    cursor.close()

    # update last_login_attempt
    formatted_date = current_time.strftime('%Y-%m-%d %H:%M:%S')
    query = "UPDATE users SET last_login_attempt = '{}' WHERE user_id = '{}'".format(formatted_date, result[0][0])
    cursor = execute_query(db_connection, query)  # run query
    cursor.close()

    #log user in
    user = User(user_id=result[0][0], username=result[0][1], password=result[0][2], email=result[0][3])
    login_user(user)
    session.permanent = True
    flash('You have been logged in!', 'success')
    next_page = request.args.get('next')
    db_connection.close() # close connection before returning
    return redirect(url_for('home'))
```

Else, the provided username and password do not match, so the ‘login_attempts’ for the user is incremented and their ‘last_login_attempt’ is updated before displaying an unsuccessful login message.

```
# else failed login attempt
else:
    # add one to login_attempts
    query = "UPDATE users SET login_attempts = '{}' WHERE user_id = '{}'".format(result[0][6] + 1, result[0][0])
    cursor = execute_query(db_connection, query)  # run query
    cursor.close()

    # update last_login_attempt
    formatted_date = current_time.strftime('%Y-%m-%d %H:%M:%S')
    query = "UPDATE users SET last_login_attempt = '{}' WHERE user_id = '{}'".format(formatted_date, result[0][0])
    cursor = execute_query(db_connection, query)  # run query
    cursor.close()

    flash('Login Unsuccessful. Please check username and password', 'danger')
    db_connection.close() # close connection before returning
    return render_template('login.html')
```

Hash passwords with salt(sensitive data)

References

General:
https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A2-Broken_Authentication
https://owasp.org/www-community/attacks/Credential_stuffing
https://www.sitelock.com/blog/owasp-top-10-broken-authentication-session-management/
https://security.googleblog.com/2015/05/new-research-some-tough-questions-for.html
Complex passwords:

Hashing:

Brute Force:
https://null-byte.wonderhowto.com/how-to/brute-force-nearly-any-website-login-with-hatch-0192225/
https://stackoverflow.com/questions/1345827/how-do-i-find-the-time-difference-between-two-datetime-objects-in-python

Password Strength:
https://www.password-depot.de/en/know-how/brute-force-attacks.htm

Session Management:
https://stackoverflow.com/questions/11783025/is-there-an-easy-way-to-make-sessions-timeout-in-flask/49891626#49891626
https://stackoverflow.com/questions/19760486/resetting-the-expiration-time-for-a-cookie-in-flask/19795394

https://flask-login.readthedocs.io/en/latest/
https://riptutorial.com/flask/example/30387/timing--out-the-login-session