COLLEGE OF ENGINEERING

Electrical Engineering and Computer Science

WHITE HAT'S TEAM:

Banks Cargill-cargillb@oregonstate.edu

Frederick Eley-eleyf@oregonstate.edu

Timothy Glew-glewt@oregonstate.edu

INTRODUCTION

- Almost everything we do in our lives involves the web, making cyber security a major concern for everyone, yet, according to Security Magazine, a website is hacked every 39 seconds.
- Our project seeks to enable students understanding of the most common web application vulnerabilities in addition to guiding them through a method of defense to encourage aspiring coders to develop security conscious habits. The web app vulnerabilities we focused on are defined by OWASP's Top Ten list.
- OWASP is an online community that produces free materials specific to the field of web application security. The Top Ten details web application security risks that represent a critical security risks to web applications.
- Every day more and more of our daily activities move to the web, a trend that has recently been accelerated by the COVID-19 pandemic forcing individuals, businesses and students alike to rapidly transition. As we make this transition, our data and sensitive information is being held by databases and web applications.
- Web design and application development is covered in depth by the required undergraduate courses for those majoring in computer science; however, without actively pursuing information about security, many beginner websites and applications suffer from major vulnerabilities, not just from malicious attacks but from user error.

Oregon State University

WHITE HAT SECURITY RESEARCH WEB APP

Deploy and secure a web application through iterative research, penetration testing, and defense guided by OWASP top ten.

How to Attack our Site:

Tautological Injection - Login

At the login page, you'll need to enter text into the username and password to bypass the form requirement. At the end of the password entry add: ' OR user_id like '%%

This will make the query read as:

```
SELECT * FROM users WHERE username = 'yourinput' AND pword =
'yourinput' OR user_id like '%%'
```

This will return all rows from the 'users' table with a user_id. Because the front end of the web application assigns the user information based on the first row, you will be logged in as if you were the first returned user_id.

This attack is based on the assumption that the attacker can guess a name for the property that holds the user's id. Since most databases we have seen use "user_id" or "id", we felt this was a safe assumption to make.

Once you have logged in and proven both that user_id is a property of the user table and the injection works, you can now play with different user_id's and log in as different users.

For example:

```
'OR user_id = '2
```

The tutorials provide specific examples of manipulating the site to demonstrate the vulnerability. This encourages users to experiment and find different ways to manipulate the site.

How to Defend our Site:

Tautological Injection - Login

For our vulnerable site, we wrote the query so that if a user knew a valid username and a valid password that would return a single row from the database, they could login. We changed this so that the database was queried solely from the username input. Since usernames must be unique, this will only return one row from the DB. We then check the user's input for their password against the DB's returned password. If they match, the user is logged in.

To further protect against users injecting queries, we created a stored procedure. A stored procedure is a type of parameterised query. A stored procedure writes the query beforehand and earmarks the locations where data will later be supplied. This clearly differentiates what is part of the query, and what is data to be used in the query, that the database understands. This prevents injected queries from being run because any query that someone attempts to inject will be interpreted as data rather than as a query.

Insecure: Dynamic Query Construction

```
query = "SELECT * FROM users WHERE `username`='{}' AND pword='{}'".format(username, password)
```

Secure: Stored Procedure

```
CREATE DEFINER=`a4dp6xjzj6ogrgmu`@`%` PROCEDURE `returnUserInfo`(IN uname varchar(20))
BEGIN
    SELECT * FROM users WHERE username = uname;
END
```

Secure: Call to Stored Procedure

```
cursor = db_connection.cursor()
cursor.callproc('returnUserInfo', [username, ])
result = cursor.fetchall()
```

Secure: Cross Validation of User Input Password and DB-stored Password

```
if result:
    #added this as validation that user input matched query results
    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)
        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'))
```

Details of the defense are provided with comparisons of the code between our insecure and secure sites.

DESCRIPTION

- We've implemented two instances of a basic to-do list web application, an insecure and secure version. Tutorials are available on the insecure site that detail each vulnerability, how a user can attack the insecure site, and how we have defended against those attacks on the secure site.
- Our sites are built with Flask, a micro web framework written in Python, and a MySQL database server. The sites are hosted through Heroku, a cloud application platform the provides free hosting services. We chose these platforms due to their general accessibility to make the findings valuable to a broad audience.
- All our source code is hosted on GitHub. While many code snippets directly related to vulnerabilities are included in our tutorials, users are encouraged to reference our codebase to increase their understanding of the vulnerabilities covered by seeing how the different pieces interact with each other.
- While there exist many online resources that discuss web application vulnerabilities and provide isolated examples, our goal with this project is to create a resource where all this information is centralized, and presented in an environment where coders who are interested in learning about web security can quickly apply what they read to see the impact.

VULNERABILITIES COVERED

- SQL Injection
- Broken Authentication
- Sensitive Data Exposure
- Broken Access Control
- Cross-Site Scripting (XSS)
- Insufficient Logging and Monitoring