Sensitive Data Exposure

White Hats: Team Members

Banks Cargill - cargillb@oregonstate.edu

Frederick Eley - eleyf@oregonstate.edu

Timothy Glew - glewt@oregonstate.edu

Description

The third web application security listed on the OWASP Top Ten Security Risks is sensitive data exposure. Without proper measures in place, sensitive data such as financial, healthcare, and personal identification information may be at risk of compromise. It is possible for this data to be stolen or modified which can lead to credit card fraud, stolen identities, unauthorized account access among other crimes. Data can be vulnerable both when being exchanged between the server and client as well as when it is stored in a database, so protection of both is imperative.

As mentioned, one of the places we need to protect sensitive data is when data is exchanged between the client and server. The primary protocol used to send data between the web client and the server is HTTP. The client can make an HTTP request to the server which is a request for the information needed to load the website. The server processes this request and then depending on the information sent in the request, sends back a reply. HTTP sends and receives information in plaintext, so attackers who can use free and widely available software can “sniff” this information and read the contents of requests and responses. This is problematic if the message contains sensitive information because there is no built-in protection. To prevent this, websites can use HTTPS to encrypt the contents of requests and responses, so that in the event messages are intercepted, the attacker will not be able to read the data. HTTPS extends HTTP by using Transport Layer Security (TLS) or Secure Sockets Layer (SSL) to implement an asymmetric keying system to encrypt communications.

Another place we need to protect sensitive data is when it is at rest in a database. One consideration is how to protect users’ passwords. It is dangerous to store passwords as plaintext, because if someone gains access to your database, they will be able to read the passwords of everyone’s account. Instead, we can make passwords more secure by storing a hash of a user’s password instead. A hash is a one-way function that takes an input and produces an output that is not human readable. A hash always produces the same output for a given input, so in this way we can hash and store a user’s password, and in the authentication process when a user enters their password we can hash their input and compare it to the stored hash value. This allows us to not have to store passwords in plaintext; however, this is not enough. It is possible for attackers to use rainbow tables, which are precomputed tables that link common passwords to their hash values. To significantly increase the difficulty for hackers to use rainbow tables or other brute force attacks we can ‘salt’ passwords before hashing them. A salt is a unique, randomly generated, string that is either prepended or appended to a password before hashing to increase the complexity of the password without requiring the user to remember this increased complexity. For example, if a user is registering with the password of

“badpassword1”, a salt of “m4k3p4ssb3tt3r” could be appended, so the combined “badpassword1m4k3p4ssb3tt3r” would be hashed and stored instead of just the password. This makes attacks with rainbow tables much more difficult, because assuming the salts are unique and randomly generated (which it was not in my example), the attacker would need an immense number of rainbow tables to precompute the hash values of all the common passwords they are testing with the unique salts of each user. This exponentially increases the space and computing resources necessary to crack every user’s password.

How to Attack our Site

Cleartext Passwords

Our insecure site stores passwords as plaintext compared to our secure site which stores the hash of salted passwords. This means that an attacker who gains access to the database of our insecure site would be able to read the usernames and passwords of registered users compared to our secure site's database where the attacker wouldn't be able to directly use the stored hashes to authenticate themself as another user.

One way to attack our insecure site would be to use SQL injection (see our SQL injection write-up for more details on how SQL injection works) to pull the usernames and passwords in our database. To do this we can create a new account or log in to an existing one, view a list, add a task to the list, and then click 'update' on a task. This will take you to a URL like the following where '1' is the list_id and '3' is the 'task_id':

```
osu-capstone-project-insecure.herokuapp.com/update_task/1/3
```

This runs the following query:

```
SELECT * FROM tasks WHERE task_id = 3
```

The above query is designed to return a single record containing the information about the selected task. This information is used to pre-fill the update form with the information of the task the user would like to edit; however, using SQL injection we could enter the following URL:

```
osu-capstone-project-insecure.herokuapp.com/update_task/1/3 UNION SELECT 99999, 1, 2, concat(username, " - ", pword), 0 FROM users WHERE user_id = 1 ORDER by task_id DESC
```

This would produce the following query:

```
SELECT * FROM tasks WHERE task_id = 3
UNION
SELECT 99999, 1, 2, CONCAT(username, " - ", pword), 0
FROM users WHERE user_id = 1
ORDER by task_id DESC
```

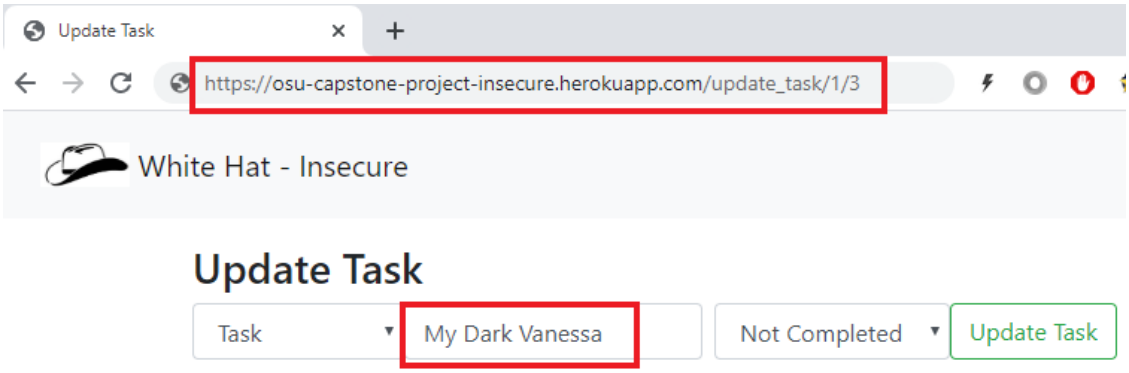
Now, in addition to pulling the task the user selected, the query uses a UNION JOIN to also return a custom record. Essentially, we are manufacturing a false 'task' record with:

- 'task_id' of 99999 (this value should be larger than any other 'task_id', so theoretically this might need to be increased if many tasks existed in the database)
- ''list_id' of 1 (which matches the real 'list_id' which is necessary for our false result to be rendered properly)
- 'dataType_id' of 2 (this can be any valid 'dataType_id')

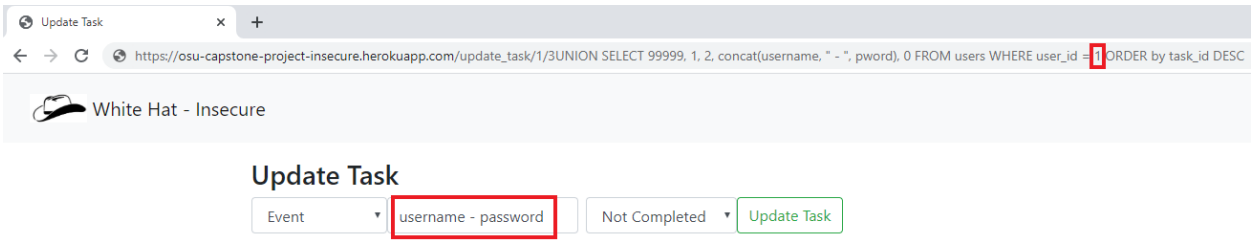
- ‘description’ set to display the username and password of the user with the ‘user_id’ specified in the WHERE clause by using CONCAT
- ‘completed’ status of 0 (this can be 1 or 0)

Our website expects a single result, so it only processes the first record returned. Thus, in our false task we set the ‘task_id’ to something presumably larger than any of the other tasks on the list (in this case we set it to 99999) so that we can ORDER the results descending by ‘task_id’ to ensure that it is processed first and rendered to the page. You can then change the parameter for ‘user_id’ in the WHERE clause to be any ‘user_id’ value to view the username and password of any registered user since passwords are stored in plaintext. Our secure site protects against SQL injection, but even if it didn’t, attackers would not be able to see the passwords of users since we only store hash values in that database.

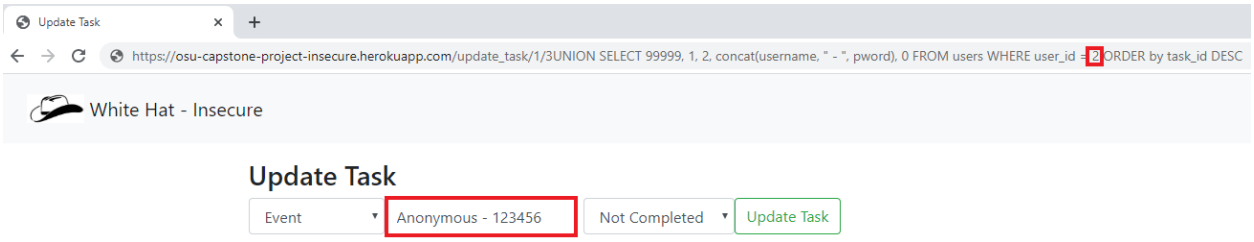
Example results without SQL injection:



Example results with SQL injection for user_id = 1:



Example results with SQL injection for user_id = 2:

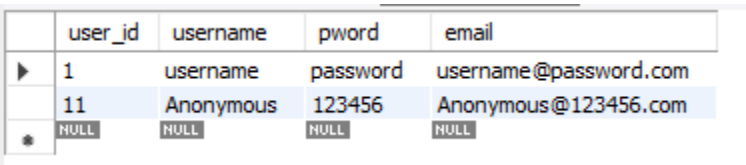


How to Defend our Site

Password Hashing

Our insecure web application stores passwords in the database in plaintext. If an attacker were able to use a SQL injection attack (see SQL Injection write-up) or other means and dump our 'users' table, they would have full access to every users' credentials. By salting users' passwords and storing the hash, if an attacker were able to access our database, they would not be able to read users' passwords and a rainbow attack would be infeasible due to salting.

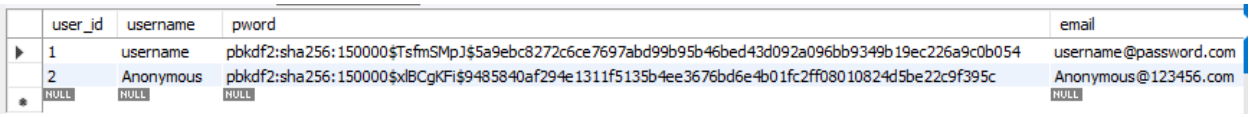
Insecure: Passwords in plaintext



user_id	username	pword	email
1	username	password	username@password.com
11	Anonymous	123456	Anonymous@123456.com
NULL	NULL	NULL	NULL

To protect this vulnerability on our secure site users' passwords are assigned a random 8-character salt, then hashed, and then it is this hash that we store along with the salt used. The screenshot below shows the same passwords as above for the same usernames after being hashed with a random salt. The value stored in the 'pword' field below includes the hash method, the salt value, and the resulting hash value in the following format: 'hash_method$salt$hash'. It is worth noting that if an attacker gains access to the database they will be able to see the salt for a user which would allow them to compute a rainbow table for a list of common passwords. While this means that a targeted attack against a single user is more feasible, the goal of storing hashed passwords is to prevent a broader attack.

Secure: Hashed passwords



user_id	username	pword	email
1	username	pbkdf2:sha256:150000$TsfmSMpJ$5a9ebc8272c6ce7697abd99b95b46bed43d092a096bb9349b19ec226a9c0b054	username@password.com
2	Anonymous	pbkdf2:sha256:150000$xlBCgKFi$9485840af294e1311f5135b4ee3676bd6e4b01fc2ff08010824d5be22c9f395c	Anonymous@123456.com
NULL	NULL	NULL	NULL

To implement salting and hashing we used the WSGI web application library Werkzeug. With Werkzeug when a user sets their password either during registration or during a password reset, we can generate a random salt that is used to hash the password and can then store the resulting hash and salt. Subsequently, when a user logs in to their account, we can use Werkzeug to compare the hash of the password they entered with their random salt which is stored in our database and compare the result to the hash value we have stored.

Insecure: user registration - no password hashing, plaintext is stored in database

```
query = ('INSERT INTO `users` '
         '(`user_id`, `username`, `pword`, `email`) '
         'VALUES (NULL, %s, %s, %s);')
data = (username, password, email)
execute_query(db_connection, query, data)
```

Secure: user registration - password is hashed with random salt before storage

```
# hash password with random 8 char salt - hash and salt are stored in hashed_password
# in the same string
hashed_password = generate_password_hash(password, salt_length=8)

cursor = db_connection.cursor()
cursor.callproc('addUser', [username, hashed_password, email, ])
```

Note that in the secure implementation in addition to storing a hashed password instead of plaintext password, we are also using a stored procedure instead of a dynamically created query string. This was done to protect against SQL injection. See our SQL injection write-up for more information. The same logic shown in the above picture is also used when updating a user's password during a password reset.

The next place we updated our secure site is when a user provides their plaintext password during login, we must compute the hash of the password provided with the salt for the user and compare this to the hash we have stored. If they match the user can be authenticated. This is different from our insecure site where we directly compare the provided plaintext password with the plaintext password stored in our database.

Insecure: login - plaintext password comparison

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

result = execute_query(db_connection, query).fetchall()  # run query
if result:
    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')
```

Secure: login - hashed password comparison

```
# else check validation that user input matched query results - successful login
elif username == result[0][1] and check_password_hash(result[0][2], password):  #
```

```
    #log user in
    user = User(user_id=result[0][0], username=result[0][1], password=result[0][2],
    login_user(user)
```

On the insecure site we take the username and plaintext provided and query for a user that matches both, and if one is found they are logged in. As mentioned previously, on the secure site we are now computing the hash of the provided password and comparing it to our stored hash. Werkzeug does this with the 'check_password_hash' method. The first parameter is a string (in our case 'result[0][2]' shown above) that contains a hashing method, a salt, and a hash value. The second parameter is a password string with which we want to hash and compare. The 'check_password_hash' method takes the salt from the first parameter, adds it to the second parameter (plaintext password), and then hashes it with the method included in the first parameter. Lastly, the method compares the result of the hash computation with the hash in the first parameter and returns true if they match, else it returns false.

Implement HTTPS

There are several ways to enforce HTTPS in web applications but the current suggestion from the Flask Security Guide is to use flask-talisman, an extension that focuses on web application security issues. The default configuration of talisman forces all connections to HTTPS unless running with debug enabled among other high priority fixes. Since we are only using it at this point for HTTPS, that is the setup covered here.

To use the default setting, we only needed to import Talisman and call it from our web application script.

```
Talisman(webapp)
```

This worked almost perfectly. When we navigated to our site, we could see that we were now protected under HTTPS but our bootstrap formatting for HTML was not being loaded or recognized. To fix this, we needed to whitelist bootstrap within Talisman's settings. We modified the content_security_policy from within our configuration file to enable pulling from self and bootstrap's content delivery network.

Modification in our configuration file:

```
TALISMAN_CSP = {'default-src': ['\'self\'', 'stackpath.bootstrapcdn.com']}
```

Modification to the Talisman initialization:

```
Talisman(webapp, content_security_policy = webapp.config['TALISMAN_CSP'])
```

Now we have full HTTPS protection without any negative impact on our site's functionality.

References

General

https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A3-Sensitive_Data_Exposure

https://www.cloudflare.com/learning/ssl/what-is-https/

https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/

https://www.cloudflare.com/learning/ssl/what-is-ssl/

https://culttt.com/2013/01/21/why-do-you-need-to-salt-and-hash-passwords/

https://en.wikipedia.org/wiki/Rainbow_table

Password Hashing

https://www.youtube.com/watch?v=jJ4awOToB6k (Password Hashing in Flask with Werkzeug)

https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/

https://werkzeug.palletsprojects.com/en/1.0.x/utils/