**ESDE Secure Assignment**

**YAPZONGXIN – P1935574 – DIT2A03**

1. **SQL INJECTIONS**

|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| **HIGH**  SQL injection is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. It happens when the attacker gives a hostile data in the input. The hostile data will then be interpreted by the vulnerable website. It generally allows an attacker to view data that they are not normally able to retrieve. This might include data belonging to other users, or any other data that the application itself is able to access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behaviour.  In some situations, an attacker can escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure or perform a denial-of-service attack. This may cause the loss of data, corruption of the web application’s database or even deny access to users. In extreme cases, it may result in a hostile takeover of the web application itself.  The snapsell web application is vulnerable to SQL Injection attacks as it allows users to inject SQL statements into the input. | **HIGH**  SQL Injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind.  Furthermore, with almost every technical advancing, hackers will be able to discover new attack vectors, and for as long as relational databases have been used in web applications, so too will SQL Injection attack vectors.  As a result, despite being the oldest form of cyber-attack, SQL Injection is still one of most common cyber-attack today. According to OWASP, one of OWASP Top 1 vulnerability is Injection. |

|  |
| --- |
| **Detailed Example** |
| A SQL Injection can be performed at the snapsell webpage in the search.html page.    The code below exposes the SQL Injection flaw where malicious user input will be treated as part of the query and it will be executed.  FirstBackEnd/models/**listings.js**    getTitle: *function* (*title*, *callback*) {  *var* dbConn = db.getConnection();      dbConn.connect(*function* (*err*) {        if (*err*) {          console.log(*err*);          return callback(*err*, null);        } else {  *var* sql = "SELECT \* FROM listings WHERE title LIKE '%" + *title* + "%'";          dbConn.query(sql, [], *function* (*err*, *result*) {            dbConn.end();            if (*err*) {              console.log(*err*)              return callback(*err*, null);            } else              return callback(null, *result*);          });        }      });    },  With this vulnerability, an attacker will be able to use and execute SQL queries as he like. For example, by executing queries using Information.Schema table into an input field, the attacker can retrieve other sensitive data stored in the database.  When the SQL statement of **1' union select 1,2,username,password,3,4,5 from snapsell.users;-- -** is entered into the search bar, the username and password of all users in the snapsell database will be displayed on the site. Hence, as such information are sensitive and should not be retrieve by non-authorised personnel, it shows that this webpage is prone to SQL Injections. |

|  |
| --- |
| **Recommendations** |
| With MySQL query() function, it is possible to auto escape the user inputs with placeholders represented by ? when the query is run. All the ? must be replaced in sequential order with values represented in an array in the same order.  This is because with placeholders, the query function will escape the input from the user such that the SQL query will search for the value as “**1' union select 1,2,username,password,3,4,5 from snapsell.users;-- -**” instead, which does not exist.  As a result, by replacing the title with ? in the statement and defining title under .query() instead, it will ensure that the input will be sanitized before the query is executed on the database server. In addition, **title = ‘%’ + title + ‘%’** is added to allow the search engine to find any values that have ‘title’ in any position (flexible search).  FirstBackEnd/models/**listings.js**    getTitle: *function* (*title*, *callback*) {  *var* dbConn = db.getConnection();      dbConn.connect(*function* (*err*) {        if (*err*) {          console.log(*err*);          return callback(*err*, null);        } else {  *title* = '%' + *title* + '%'  *var* sql = "SELECT \* FROM listings WHERE title LIKE ?";          dbConn.query(sql, [*title*], *function* (*err*, *result*) {            dbConn.end();            if (*err*) {              console.log(*err*)              return callback(*err*, null);            } else              return callback(null, *result*);          });        }      });    },  After changing the codes, the result now will be that users will no longer be able to induce SQL Injection into the input fields to retrieve sensitive or other data/tables in the database. It will display an empty result since there is no such listings in the database called (1' union select 1,2,username,password,3,4,5 from snapsell.users;-- -). In addition, users will still be able to have flexibilities when they search for listings. |

1. **Cross Site Scripting (XSS) Attack**

|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| **HIGH**  Cross-site scripting, also known as XSS, is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It is a type of injection, in which malicious scripts are injected into trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.  XSS is split into two categories, non-persistent and persistent.  For non-persistent XSS, also known as reflected XSS, the malicious code is executed by a victim’s browser, and the payload is not stored anywhere. It is returned as part of the response from the HTML that a server sends.  On the other hand, persistent XSS attack occurs when a web application stores user input and later serves it to others. The malicious code is executed by a victim’s browser, and the payload is stored on the victim server. It is returned as part of the response from the HTML that a server sends. | **HIGH**  The likelihood of XSS attacks are considerably high due to the availability of automated exploitation tools on the internet. This increases the chances of XSS attacks due to the ease of obtaining such applications, which makes attacks so much easier. With XSS there is an added factor. The attacker commits these crimes on websites that, in the user’s eye, should be fully trustworthy. Hence, users may not be aware and can easily fall into XSS attacks.  According to PandaSecurity, in the last nine years, the most frequent bug on websites the world has, had been the vulnerability of XSS (Cross-site Scripting), which makes up 18% of the bugs found. |

|  |
| --- |
| **Detailed Example** |
| The XSS attack can be performed in the product.html, on the page where users add a product to their listing.    So, every time the users wants to view their listings by going to home.html, the script tag will run and an alert message will pop up.    The XSS attack can take place as there are no input validation in the app.js file. Furthermore, every time the page (home.html) was reloaded, the alert message will pop up. This is because the website is vulnerable to persistent XSS attacks. The code below shows that there is no input validation on the server-side. Hence, malicious input written in product.html can be stored into the database and run every time users enter the home.html page.  FirstBackEnd/controller/**app.js**  app.post('/listings/', *function* (*req*, *res*) {  *var* title = *req*.body.title;  *var* description = *req*.body.description;  *var* price = *req*.body.price;  *var* poster\_id = *req*.body.fk\_poster\_id;  *var* image = *req*.body.image;    listings.insertListings(title, description, price,image, poster\_id, *function* (*err*, *result*) {      if (*err*) {  *res*.status = 500;  *res*.send(`{"Internal Server Error"}`);      } else {  *res*.status(201);  *res*.send({ "listingID": *result*.insertId });      }    });  });  By confirming that this webpage is susceptible to XSS attacks, it suggest that attackers have the power to run a much more dangerous codes rather than this script tag <script>alert(‘alert message’)</script> in the website. Attackers may use this vulnerability to run some codes and steal personal data from the users. |

|  |
| --- |
| **Recommendations** |
| **Manual Validation (RegExp)**  To prevent the ability for XSS attacks at the product.html page, we can introduce input validation for the input fields. Currently, any data that is entered and submitted will be sent to the server as JSON object which will then be stored in the database. To ensure that only appropriate data is being entered, we can fix a format or limit characters in the input field. One way to do so is to use RegExp.  By using RegExp, the server will be able to check for the contents of the input. To prevent attackers from submitting malicious input, we can filter out special characters such as quotation marks, slashes or arrow brackets. We can do so by limiting the users to only uppercase, lowercase, numbers and some special characters where only whitelisted characters should be allowed and special HTML characters should certainly not be allowed.  FirstBackEnd/controller/**app.js**  app.post('/listings/', *function* (*req*, *res*) {  *var* title = *req*.body.title;  *var* description = *req*.body.description;  *var* price = *req*.body.price;  *var* poster\_id = *req*.body.fk\_poster\_id;  *var* image = *req*.body.image;  *var* validation = new *RegExp*('^[a-zA-Z0-9\s,]\*$')    if(validation.test(title) && validation.test(description) && validation.test(price) && validation.test(image)){      listings.insertListings(title, description, price,image, poster\_id, *function* (*err*, *result*) {        if (*err*) {  *res*.status = 500;  *res*.send(`{"Internal Server Error"}`);        } else {  *res*.status(201);  *res*.send({ "listingID": *result*.insertId });        }      });    } else{  *res*.status = 500;  *res*.send(`{"Internal Server Error"}`);    }  });  This will ensure that the information in the input field will not go through and instead, an error message will pop up.    **Built-in Validation (Libraries)**  Another way to prevent XSS attacks is by using external Nodejs validator library. This library helps you to validate and sanitize strings in your application. Common functions are validating emails, credit cards, urls etc which are already defined in the library and you can use it like an API. This can be done by simply installing validator, **npm install validator --save**, and writing some codes to call the functions. Since node validator is installed, one can use validator.escape() to convert special character into HTML entities.  FirstBackEnd/validation/**validate.js**  *const* validator = require('validator');  *var* validate={          sanitizing:*function*(*result*){                for (i=0; i < *result*.length; i++){  *var* row= *result*[i]                  console.log(row);                  for(*var* key in row){                      val = row[key];                      if (typeof val === "string"){                          row[key] = validator.escape(val);                      }                  }              }          }  };  module.exports = validate;  Characters such as <,>,&,’,”,/ should be converted to the corresponding HTML entities, so that they won’t be interpreted as code like JavaScript in XSS attacks.  FirstBackEnd/controller/**app.js**  *var* validate = require('../validation/validate');  app.get('/users/:user\_id/listings/', *function* (*req*, *res*) {  *var* user\_id = *req*.params.user\_id;      listings.getListingsByUsers(user\_id, *function* (*err*, *result*) {      if (*err*) {  *res*.status = 500;  *res*.send(null);      } else {  *res*.status(200);        validate.sanitizing(*result*);  *res*.send(*result*);      }    });  });  As such, all the data will be processed by the function called sanitizing() to sanitize the information being passed into HTML entities. For example, the alert script tag will no longer run since it is being treated as HTML instead, hence there will be no alert message. |

1. **Broken Authentication**

|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| **HIGH**  Broken authentication is typically caused by poorly implemented authentication and session management functions. Broken authentication attacks aim to take over one or more accounts giving the attacker the same privileges as the attacked user. Authentication is “broken” when attackers are able to compromise passwords, keys or session tokens, user account information, and other details to assume user identities.  Due to poor design and implementation of identity and access controls, the prevalence of broken authentication is widespread. One such example of this kind of attack is due to predictable login credentials, where the application permits default, weak, or well-known passwords, such as “Password1” or “admin/admin“.  Attackers have to gain access to only a few accounts, or just one admin account to compromise the system. Depending on the domain of the application, this may allow money laundering, social security fraud, and identity theft, or disclose legally protected highly sensitive information. | **MEDIUM**  The prevalence of broken authentication is widespread due to the design and implementation of most identity and access controls. Session management is the bedrock of authentication and access controls, and is present in all stateful applications.  Attackers can detect broken authentication using manual means and exploit them using automated tools with password lists and dictionary attacks. However, as attackers will require knowledge to be able to use automated tools and they need to go through tedious process pf guessing the passwords, the likelihood of brute force method will not be as high as XSS or Injections. |

|  |
| --- |
| **Detailed Example** |
| By allowing and permitting any weak or common password without validation, attackers will be able to use a list of the most common passwords (such as ‘password’ and ‘123456789’) and try each one in turn – again using automated scripts. This can be done by brute forcing password or through credential stuffing, which is the use of automated tools to test a list of valid usernames and passwords, stolen from one company, against the website of another company.  According to register.html in snapsell webpage, users are able to register an account with any username or password such as username 123 and password 123.    The username and password of the registered account was successfully stored in MySQL database as shown below.    This situation of allowing weak and common username and password happened because there are no checking and validating of the length and complexity of the password. As a result, this may allow attackers to easily steal the weak passwords of those users.  FirstFrontEnd/public/**register.html**      <button class="btn btn-lg btn-primary btn-block" type="submit" id="register">Register</button> |

|  |
| --- |
| **Recommendations** |
| To ensure that only strong password of minimum length is used to register the account, we can validate and prompt users to create a password that is strong. This will in turn create difficulties for the attackers to guess the passwords of the users since it is a lot harder to predict them. This can be done by adding an onclick function inside register button to validate the password.  FirstFrontEnd/public/**register.html**    <form id="form-register" name="registerForm" form class="form-signin">      <h1 class="h3 mb-3 font-weight-normal">Register</h1>      <label for="inputEmail" class="sr-only">Username</label>      <input type="text" id="username" class="form-control" required placeholder="Username" required autofocus>      <p></p>      <label for="inputpic" class="sr-only">Profile URL Pic</label>      <input type="text" id="pic" class="form-control" placeholder="Profile URL Pic" required autofocus>      <p></p>      <label for="inputPassword" class="sr-only">Password</label>      <input type="password" name='registerPassword' id="password" required class="form-control" placeholder="Password" required>      <button class="btn btn-lg btn-primary btn-block" type="submit" id="register" onclick="validatepassword(document.registerForm.registerPassword)">Register</button>    </form>  To do so, a function called passwordValidation can be created to check and validate the password of the users. The regex helps to check if the password has minimum of eight characters, at least one uppercase letter, one lowercase letter, one number and one special character. If the password does not pass the validation check, an alert message will pop up in register.html, alerting users to check their password.  FirstFrontEnd/public/**register.html**    <script>  *function* validatepassword(*input*){  *var* password= /^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)(?=.\*[@$!%\*?&])[A-Za-z\d@$!%\*?&]{8,}$/        if (*input*.value.match(password)) {        $(document).ready(*function* () {  *const* baseUrl = "http://localhost:8081";          $("#form-register").submit((*event*) *=>* {  *event*.preventDefault()  *const* username = $("#username").val();  *const* pic = $("#pic").val();  *const* password = $("#password").val();            axios.post(`${baseUrl}/users/`, {              username: username,              profile\_pic\_url: pic,              password: password            })              .then((*response*) *=>* {                console.log('Successfully Registered!')                window.location.href = "http://localhost:3001/login.html";              })              .catch((*error*) *=>* {                console.log("Error")              })          })        })      }else {        alert("Minimum eight characters, at least one uppercase letter, one lowercase letter, one number and one special character")      }    }    </script> |

1. **Sensitive Data Exposure**

|  |  |
| --- | --- |
| **Vulnerability/Risk** | **Likelihood** |
| **MEDIUM**  Sensitive data exposure occurs when an application, company, or other entity inadvertently exposes personal data. It occurs as a result of not adequately protecting a database where information is stored. This might be a result of a multitude of things such as weak encryption, no encryption or software flaws.  This snapsell website has its password exposed as they are not encrypted, salted and hashed. As such, they are prone to attacks since sensitive information like password are not encrypted and instead, are shown as plain text that can be easily retrieve and read by attackers. | **MEDIUM**  Over the last few years, this has been the most common impactful attack. The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm, protocol and cipher usage is common, particularly for weak password hashing storage techniques. For data in transit, server-side weaknesses are mainly easy to detect, but hard for data at rest.  When building an application, many are going to down-prioritise protection of sensitive data, and even if the developer is aware of the fact that they should, for example, hash passwords, it is common to plan to do this afterwards. A workable application is the top priority and once the application is working, the planned protection is forgotten, or simply skipped.  Dealing with crypto is also one of the most difficult things to do. It is therefore common to make mistakes when implementing a self-built solution, which will result in insufficient protection of data.  Sensitive Data Exposure is therefore a typical vulnerability that is worst for small players, like hobby projects and smaller companies. However, as can be seen by looking at some well-known events, big players are affected by these vulnerabilities as well, but not as often. |

|  |
| --- |
| **Detailed Example** |
| This vulnerability in snapsell is due to the password not being hashed and encrypted when a user creates an account. This allows the attacker to use applications such as burp proxy to steal and read the username and password since they are not properly hashed. As a result, attackers can retrieve the credentials of a user and log into their accounts.  According to the MySQL database, the password of the users is not hashed and encrypted. This allow attackers and those managing the database to be able to see and read the data under the password columns.  **(Seen by authorised personnel)**    **(Seen by attackers)**    According to the codes that manages and store the user’s password after password had been created, there are no proper techniques and codes to hash and encrypt the password, leaving the password open and exposed for easy identification and reading.  FirstBackEnd/model/**users.js**    insertUsers: *function* (*username*, *profile\_pic\_url*, *password*, *callback*) {  *var* dbconn = db.getConnection();  dbconn.connect(*function* (*err*) {        if (*err*) {          console.log(*err*);          return callback(*err*, null);        } else {          console.log(“Connected!”);          sqlStr = `Insert into users(username,profile\_pic\_url,password) values(‘${*username*}’,’${*profile\_pic\_url*}’,’${*password*}’)`;          dbconn.query(sqlStr, [], *function* (*err*, *result*) {            dbconn.end();            if (*err*) {              console.log(*err*);              return callback(*err*, null);            } else {              return callback(null, *result*);            }          });        }  });    }, |

|  |
| --- |
| **Recommendations** |
| To prevent the sensitive password from being exposed and readable by attackers, we can use hashing to make the password non-readable. Byhashing the password when the frontend of the application sends the data to the backend of the application, the backend application will encrypt the data with a secret key before storing the data into the database. This will allow sensitive information like password to be protected against attackers.  FirstBackEnd/models/**users.js**  *const* saltRounds = 10;    insertUsers: *function* (*username*, *profile\_pic\_url*, *password*, *callback*) {  *var* dbconn = db.getConnection();      dbconn.connect(*function* (*err*) {        if (*err*) {          console.log(*err*);          return callback(*err*, null);        } else {          bcrypt.hash(*password*, saltRounds, *function*(*err*, *hash*) {            if(*err*){                    console.log(*err*);                    return callback(*err*,null);            }  *password* = *hash*;          console.log("Connected!");          sqlStr = `Insert into users(username,profile\_pic\_url,password) values('${*username*}','${*profile\_pic\_url*}','${*password*}')`;          dbconn.query(sqlStr, [], *function* (*err*, *result*) {            dbconn.end();            if (*err*) {              console.log(*err*);              return callback(*err*, null);            } else {              return callback(null, *result*);            }          });          });        }      });    },      The password in the database should now be hashed after the codes above are added. However, as the password in the database was converted into encrypted format, users will not be able to login now. This is because even though the username and password are correctly inputted by the user, as the password has been hashed, the codes will compare the hashed password against the username instead, which the users will not be able to know the hash format of their password. As a result, to ensure that the password inputted by the users when they are trying to login is converted to the hashed password so that the password and username matches, we will have to modify the codes in the login function.  FirstBackEnd/model/**users.js**    loginUser: *function* (*username*, *password*, *callback*) {  *var* dbconn = db.getConnection();      dbconn.connect(*function* (*err*) {        if (*err*) {          console.log(*err*);          return callback(*err*, null);        }        else {          console.log("Connected!");  *var* sql = `select \* from users where username='${*username*}'`;          dbconn.query(sql, [], *function* (*err*, *result*) {            dbconn.end();            if (*err*) {              console.log("Err: " + *err*);              return callback(*err*, null, null);            } else {  *var* token = "";  *var* i;              if (*result*.length == 1) {                hash=*result*[0].password;                bcrypt.compare(*password*,hash,*function*(*err*,*res*){                  if(*res*){                      token = jwt.sign({ id: *result*[0].id }, config.key, {                        expiresIn: 86400 //expires in 24 hrs                      });                      console.log("@@token " + token);                      return callback(null, token, *result*);                    } //if(res)                    else {                      console.log("Username or Password does not match");  *var* err2 = new *Error*("Username or Password does not match.");                      err2.statusCode = 404;                      console.log(err2);                      return callback(err2, null, null);                    }                });            } else{  *var* err2 = new *Error*("UserID/Password does not match.");                err2.statusCode=500;                return callback(err2,null,null);            }          }          });        }      });    },  **(success)** |

# **References**

Mark Gorrie, S. D.–A. (n.d.). *Sensitive data exposure: What is it and how it’s different from a data breach*. Retrieved from Norton: https://us.norton.com/internetsecurity-privacy-sensitive-data-exposure-how-its-different-from-data-breach.html

OWASP. (2020). *OWASP*. Retrieved from OWASP: https://owasp.org/

Series, A. S. (2018, June 21). *OWASP Top 10: Broken Authentication Security Vulnerability Practical Overview*. Retrieved from ImmuniWeb: https://www.immuniweb.com/blog/OWASP-broken-authentication-attack.html#:~:text=update%20this%20page.-,Broken%20authentication,of%20user%20accounts%20being%20breached.

Team, N. S. (2019, April 18). *The Cross-site Scripting (XSS) Vulnerability: Definition and Prevention*. Retrieved from NetSparker: https://www.netsparker.com/blog/web-security/cross-site-scripting-xss/#:~:text=It%20ranges%20from%20user's%20Session,and%20take%20over%20the%20account.