Fix Mobile App Vulnerabilities for the Hackazon Application

Hayden Eubanks

School of Business, Liberty University

CSIS 486-D01

Prof. Backherms

December 10, 2023

Fix Mobile App Vulnerabilities for the Hackazon Application

Introduction:

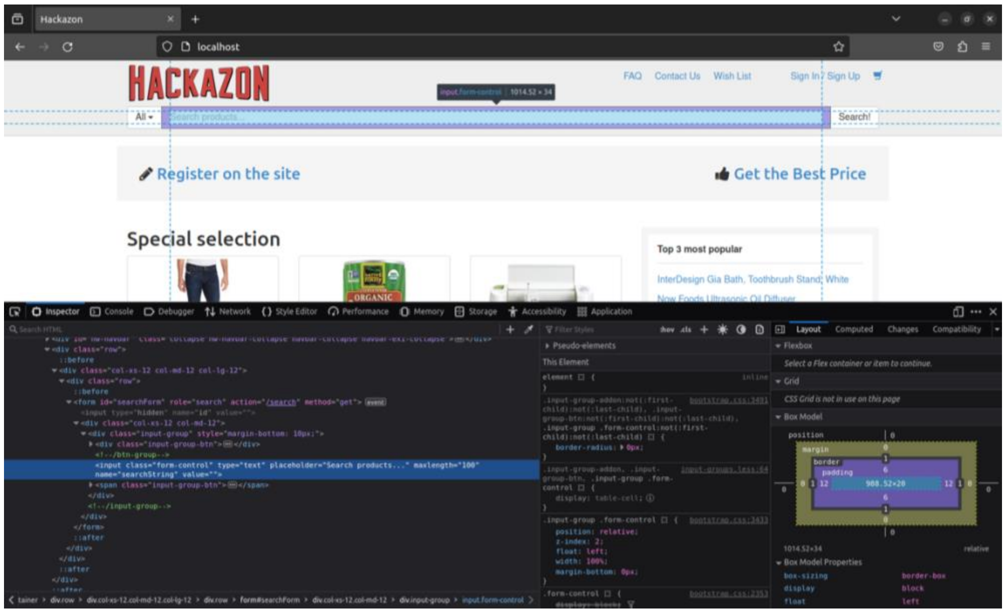
One of the most common vulnerabilities discovered within web applications is vulnerability to SQL injection (OWASP, 2021), and as such it is essential for security professionals to be able to detect this vulnerability and implement mitigation strategies within the source code for the application. SQL injection (SQLi) is a form of attack where a malicious actor injects SQL code to get the webserver to behave in a way other than what it is intended to (Simpson & Anthill, 2017). For example, malicious code may be injected to modify, insert, or destroy records in a database or to have the database return values that would otherwise be outside of the scope of the query. This is often accomplished by prematurely exiting a SQL query through the insertion of quotation or comment characters and then inserting malicious code to be passed as part of the query (PortSwigger, 2023). The data held within databases can serve as a treasure trove for attackers and as SQLi vulnerabilities deal directly with the database it can easily be identified as an attractive vector of attack. For this reason, security professionals must understand how SQLi vulnerabilities occur and how they can be mitigated to stop malicious actors and protect the underlying data.

Despite the severity of SQLi vulnerabilities, they are relatively easy to mitigate, and through examining mitigation strategies a security professional can be better equipped to address SQL injection within their web applications. To inject malicious code or prematurely end a SQL query, special characters such as quotations or characters used in SQL comments must be used (IBM, 2023). As these special characters enable SQLi to be performed, the injection can then be mitigated through the escaping or replacement of these characters before they are sent to the database for processing (OWASP, 2023). However, the characters that need to be replaced as well as the correct processing to be performed to clean input varies depending on the configuration of the database including factors such as the type of SQL used (Microsoft, 2023). For this reason, it is better practice to utilize prepared statements where the statement is preconfigured and the input then passed through a manner where it cannot alter the overall structure of the query (PHP, n.d.). Prepared statements are a powerful tool for developers of web applications and through utilizing prepared statements it can be better ensured that data passed to the application will not result in the execution of injected SQL code.

Throughout the Hackazon application, several vulnerabilities to SQL injection exist including vulnerabilities to both standard and blind SQLi attacks. Blind SQLi attacks differ in that they do not directly seek to execute complex queries but pass Boolean values that should return true or false to gain a greater understanding of the structure of the underlying query (OWASP, 2023). These SQLi vulnerabilities were discovered and discussed in a previous report and the objective of this report is to outline and discuss how these vulnerabilities can be mitigated within the source code of the Hackazon web application. This analysis will cover input filtering through methods such as addslashes or my_sql_real_escape_string as well as how these methodologies may fail to fully cover a SQLi vulnerability. Following this, an examination of prepared statements will be performed including an explanation of how prepared statements can better mitigate SQLi vulnerabilities and increase the control of the SQL query passed to the database. Prepared statements allow a developer to craft a SQL query before it is passed to a database limiting the ability for a malicious actor to alter the query structure and as such have been accepted as a best practice for managing dynamic input for SQL queries (Microsoft, 2023). Examining SQLi vulnerabilities within the Hackazon application can then allow a security professional to better understand mitigation strategies and effectively employ SQLi mitigation in applications they oversee.

Before and After for Code Segments:

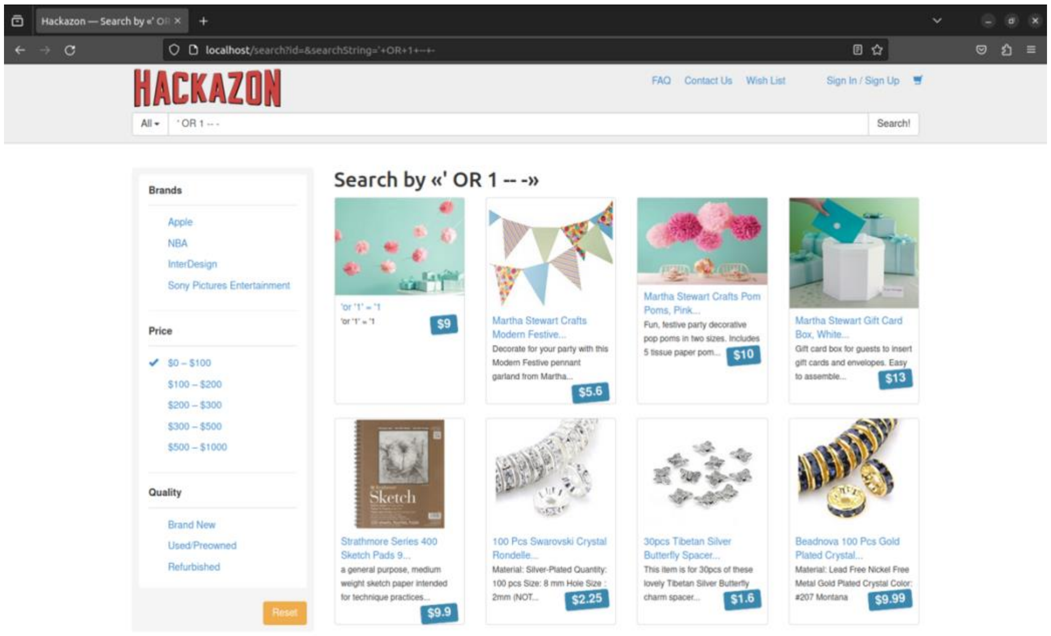
Screenshot 1: Identification of Vulnerable Search Form



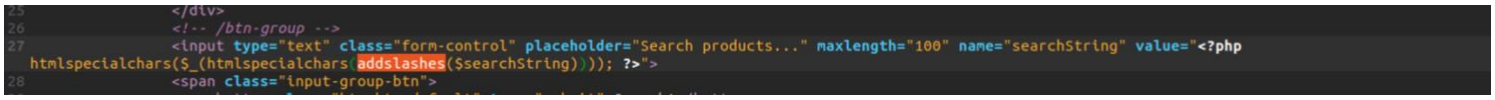
```
▾<div class="row">
   ::before
  ▾<div class="col-xs-12 col-md-12 col-lg-12">
    ▾<div class="row">
       ::before
      ▾<form id="searchForm" role="search" action="/search" method="get"> event
         <input type="hidden" name="id" value="">
        ▾<div class="col-xs-12 col-md-12">
          ▾<div class="input-group" style="margin-bottom: 10px;">
            ▸<div class="input-group-btn">…</div>
             <!--/btn-group-->
             <input class="form-control" type="text" placeholder="Search products..." maxlength="100"
             name="searchString" value="">
            ▸<span class="input-group-btn">…</span>
           </div>
           <!--/input-group-->
         </div>
       </form>
       ::after
     </div>
   </div>
   ::after
 </div>
```

```
tainer › div.row › div.col-xs-12.col-md-12.col-lg-12 › div.row › form#searchForm › div.col-xs-12.col-md-12 › div.input-group › input.form-control
```

Screenshot 2: Proof of Vulnerable Search Form

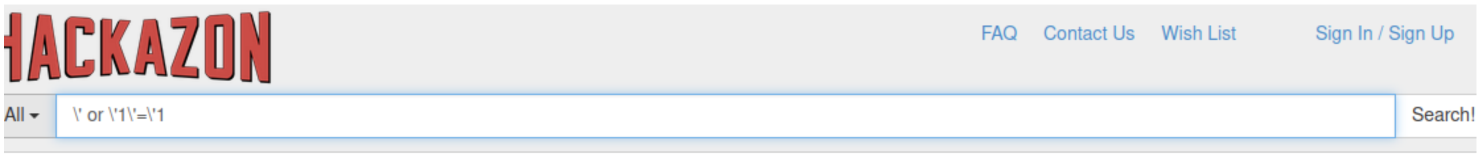

Hackazon — Search by «' OR
localhost/search?id=&searchString='+OR+1+--+-
FAQ
Contact Us
Wish List
Sign In / Sign Up
All
' OR 1 -- -
Search!
Search by «' OR 1 -- -»
Brands
Apple
NBA
InterDesign
Sony Pictures Entertainment
Price
$0 – $100
$100 – $200
$200 – $300
$300 – $500
$500 – $1000
Quality
Brand New
Used/Preowned
Refurbished
Reset
'or '1' = '1
$9
Martha Stewart Crafts Modern Festive...
$5.6
Martha Stewart Crafts Pom Poms, Pink...
$10
Martha Stewart Gift Card Box, White...
$13
Strathmore Series 400 Sketch Pads 9...
$9.9
100 Pcs Swarovski Crystal Rondelle...
$2.25
30pcs Tibetan Silver Butterfly Spacer...
$1.6
Beadnova 100 Pcs Gold Plated Crystal...
$9.99

Screenshot 3: Vulnerable Code For Search Field With addslashes Function Added



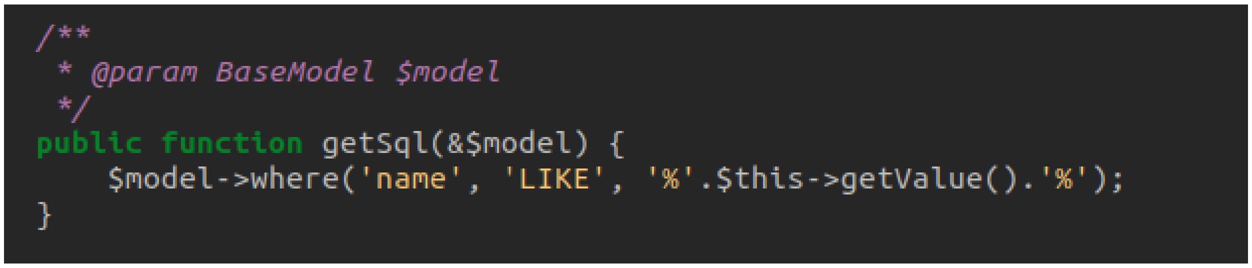
```
                </div>
                <!-- /btn-group -->
                <input type="text" class="form-control" placeholder="Search products..." maxlength="100" name="searchString" value="<?php
htmlspecialchars($_(htmlspecialchars(addslashes($searchString)))); ?>">
                <span class="input-group-btn">
```

Screenshot 4: Result of addslashes Encoding



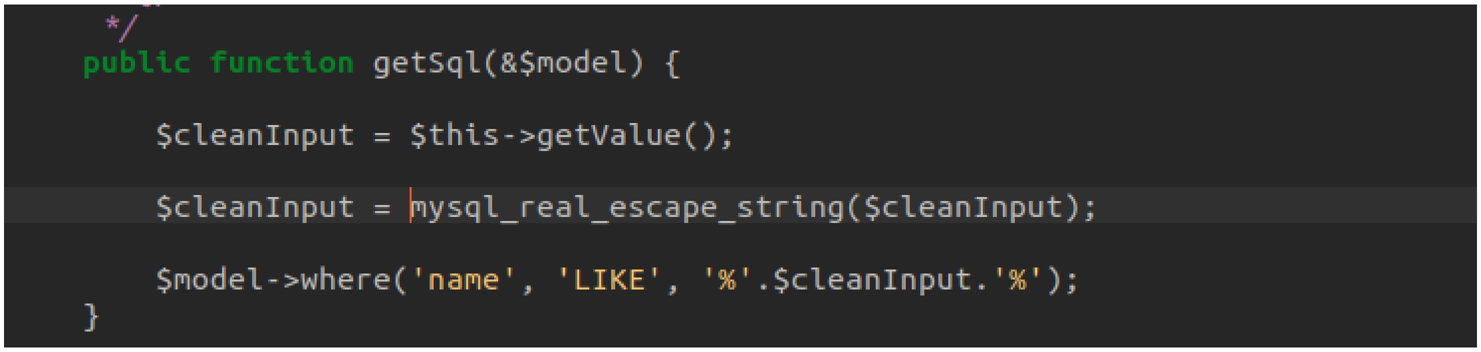
```
\' or \'1\'=\'1
```

Screenshot 5: Function Containing Part of SQL Query Generation Used In Search



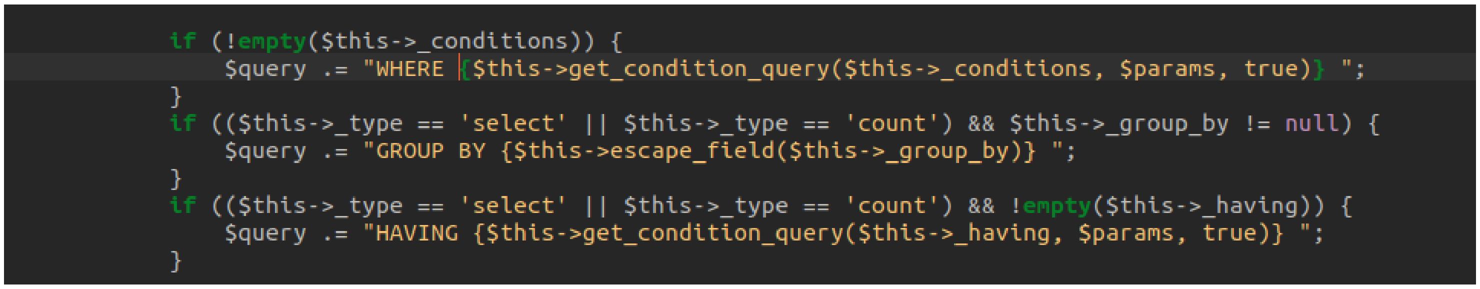
```
/**
 * @param BaseModel $model
 */
public function getSql(&$model) {
    $model->where('name', 'LIKE', '%'.$this->getValue().'%');
}
```

Screenshot 6: Attempt to First Clean Input before it is used in SQL Query



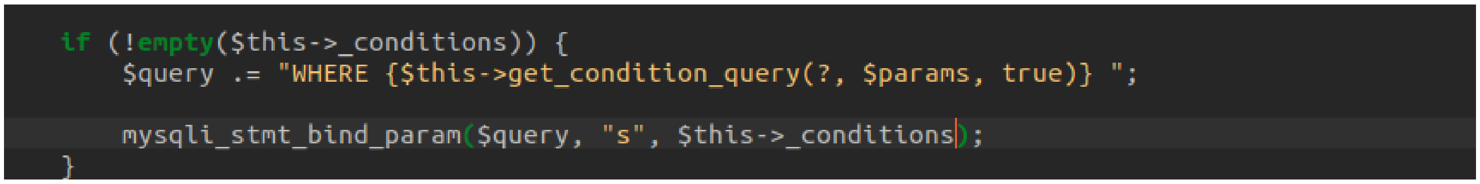
```
 */
public function getSql(&$model) {

    $cleanInput = $this->getValue();

    $cleanInput = mysql_real_escape_string($cleanInput);

    $model->where('name', 'LIKE', '%'.$cleanInput.'%');
}
```

Screenshot 7: Additional Section of SQL Query Generator



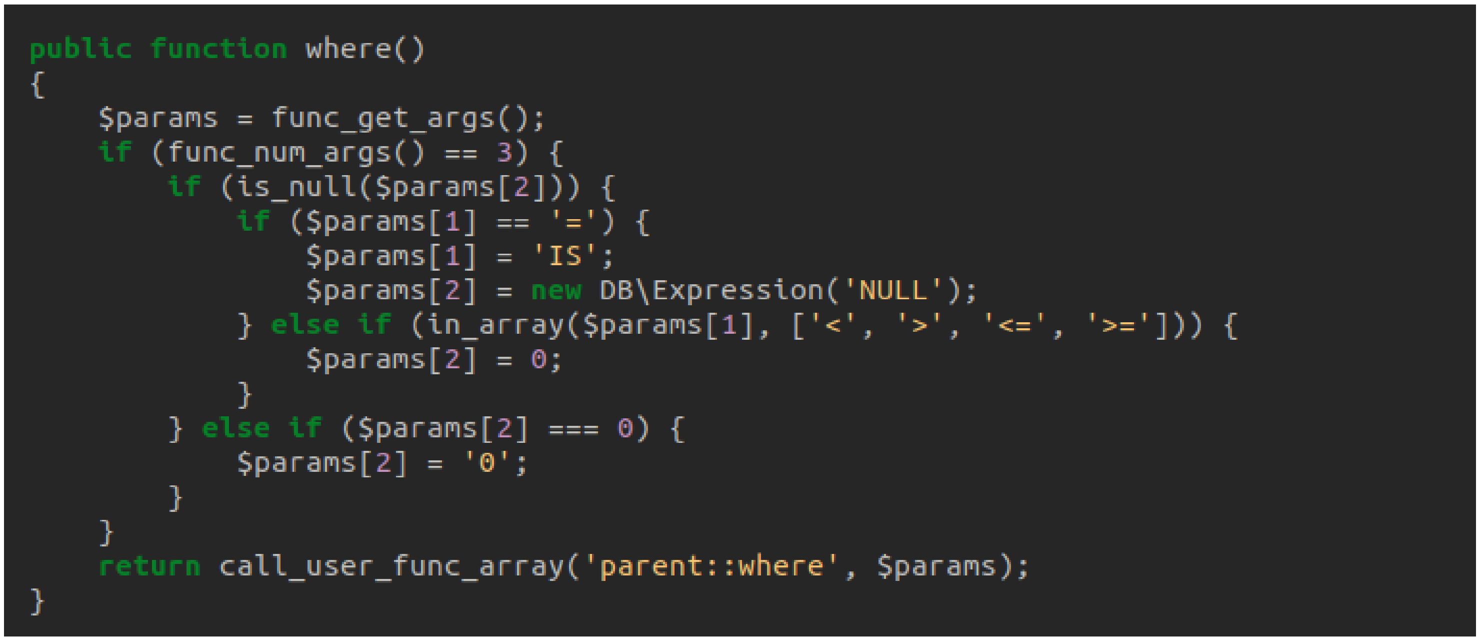
```
if (!empty($this->_conditions)) {
    $query .= "WHERE {$this->get_condition_query($this->_conditions, $params, true)} ";
}
if (($this->_type == 'select' || $this->_type == 'count') && $this->_group_by != null) {
    $query .= "GROUP BY {$this->escape_field($this->_group_by)} ";
}
if (($this->_type == 'select' || $this->_type == 'count') && !empty($this->_having)) {
    $query .= "HAVING {$this->get_condition_query($this->_having, $params, true)} ";
}
```

Screenshot 8: Creating a Prepared Statement During Query Generation



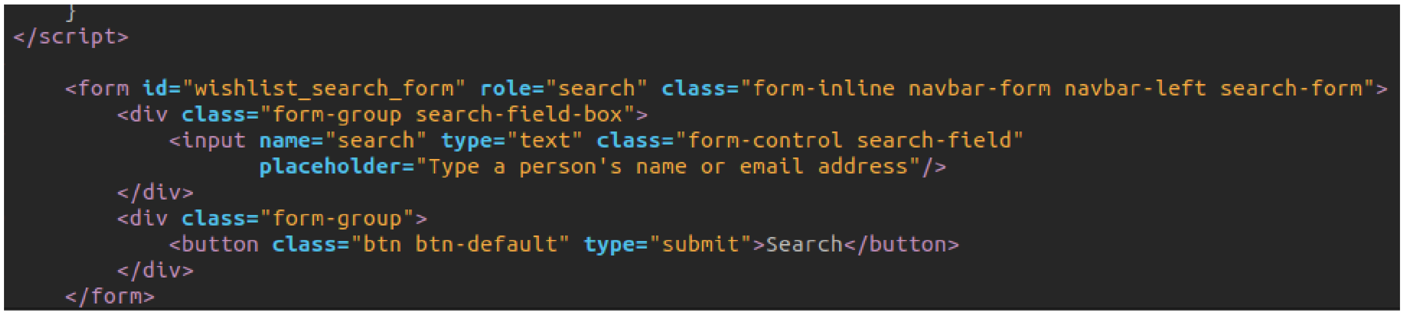
```
if (!empty($this->_conditions)) {
    $query .= "WHERE {$this->get_condition_query(?, $params, true)} ";

    mysqli_stmt_bind_param($query, "s", $this->_conditions);
}
```

Screenshot 9: Remaining SQL Query Generator When Where Filter Is Used



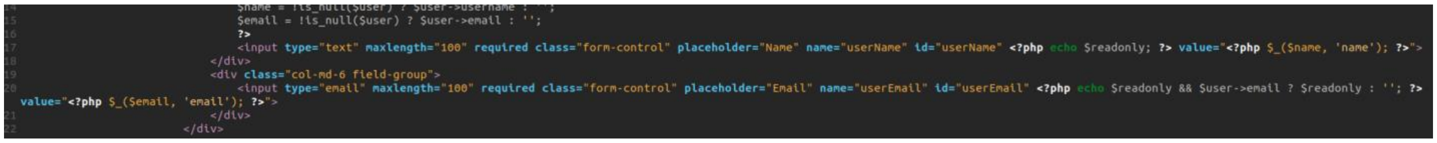
```
public function where()
{
    $params = func_get_args();
    if (func_num_args() == 3) {
        if (is_null($params[2])) {
            if ($params[1] == '=') {
                $params[1] = 'IS';
                $params[2] = new DB\Expression('NULL');
            } else if (in_array($params[1], ['<', '>', '<=', '>='])) {
                $params[2] = 0;
            }
        } else if ($params[2] === 0) {
            $params[2] = '0';
        }
    }
    return call_user_func_array('parent::where', $params);
}
```

Screenshot 10: Vulnerable Wishlist Search Form



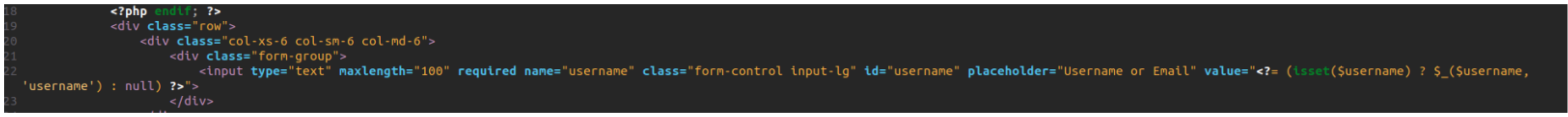
```
    }
</script>

    <form id="wishlist_search_form" role="search" class="form-inline navbar-form navbar-left search-form">
        <div class="form-group search-field-box">
            <input name="search" type="text" class="form-control search-field"
                   placeholder="Type a person's name or email address"/>
        </div>
        <div class="form-group">
            <button class="btn btn-default" type="submit">Search</button>
        </div>
    </form>
```

Screenshot 11: Vulnerable Review Form



```
                    $name = !is_null($user) ? $user->username : '';
                    $email = !is_null($user) ? $user->email : '';
                    ?>
                    <input type="text" maxlength="100" required class="form-control" placeholder="Name" name="userName" id="userName" <?php echo $readonly; ?> value="<?php $_($name, 'name'); ?>">
                </div>
                <div class="col-md-6 field-group">
                    <input type="email" maxlength="100" required class="form-control" placeholder="Email" name="userEmail" id="userEmail" <?php echo $readonly && $user->email ? $readonly : ''; ?>
value="<?php $_($email, 'email'); ?>">
                </div>
            </div>
```

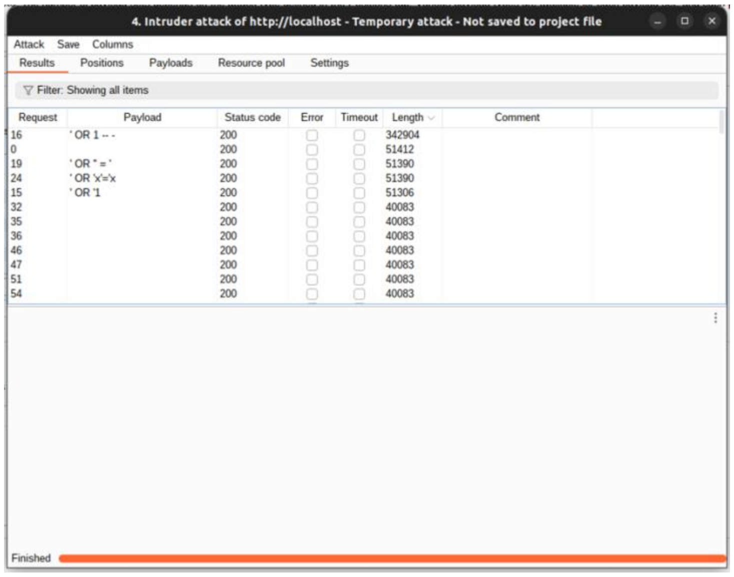
Screenshot 12: Vulnerable Sign-In Form



```
            <?php endif; ?>
            <div class="row">
                <div class="col-xs-6 col-sm-6 col-md-6">
                    <div class="form-group">
                        <input type="text" maxlength="100" required name="username" class="form-control input-lg" id="username" placeholder="Username or Email" value="<?= (isset($username) ? $_($username,
'username') : null) ?>">
                    </div>
```

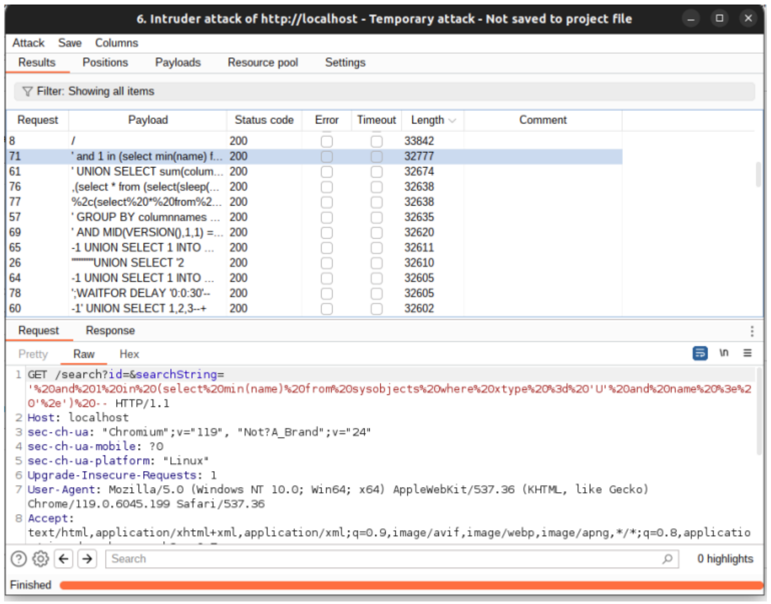
Evidence of Fuzzing:

Screenshot 13: Initial Fuzzing Showing Vulnerabilities


4. Intruder attack of http://localhost - Temporary attack - Not saved to project file
Attack Save Columns
Results Positions Payloads Resource pool Settings
Filter: Showing all items
Finished

Request	Payload	Status code	Error	Timeout	Length	Comment
16	' OR 1 -- -	200	☐	☐	342904	
0		200	☐	☐	51412	
19	' OR " = '	200	☐	☐	51390	
24	' OR 'x'='x	200	☐	☐	51390	
15	' OR '1	200	☐	☐	51306	
32		200	☐	☐	40083	
35		200	☐	☐	40083	
36		200	☐	☐	40083	
46		200	☐	☐	40083	
47		200	☐	☐	40083	
51		200	☐	☐	40083	
54		200	☐	☐	40083	

Screenshot 14: Fuzzing Showing No Vulnerability


6. Intruder attack of http://localhost - Temporary attack - Not saved to project file
Attack Save Columns
Results Positions Payloads Resource pool Settings
Filter: Showing all items
Request Response
Pretty Raw Hex
Search
0 highlights
Finished

Request	Payload	Status code	Error	Timeout	Length	Comment
8	/	200	☐	☐	33842	
71	' and 1 in (select min(name) f...	200	☐	☐	32777	
61	' UNION SELECT sum(colum...	200	☐	☐	32674	
76	,(select * from (select(sleep(...	200	☐	☐	32638	
77	%2c(select%20*%20from%2...	200	☐	☐	32638	
57	' GROUP BY columnnames ...	200	☐	☐	32635	
69	' AND MID(VERSION(),1,1) =...	200	☐	☐	32620	
65	-1 UNION SELECT 1 INTO ...	200	☐	☐	32611	
26	"""""""UNION SELECT '2	200	☐	☐	32610	
64	-1 UNION SELECT 1 INTO ...	200	☐	☐	32605	
78	';WAITFOR DELAY '0:0:30'--	200	☐	☐	32605	
60	-1' UNION SELECT 1,2,3--+	200	☐	☐	32602	

```
GET /search?id=&searchString=
'%20and%201%20in%20(select%20min(name)%20from%20sysobjects%20where%20xtype%20%3d%20'U'%20and%20name%20%3e%2
0'%2e')%20-- HTTP/1.1
Host: localhost
sec-ch-ua: "Chromium";v="119", "Not?A_Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Linux"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/119.0.6045.199 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,applicatio
```

Discussion of SQLi Vulnerabilities:

Several vulnerabilities to SQL injection can be observed throughout the Hackazon application (Screenshot 13) and through examining mitigation strategies for these vulnerabilities a greater understanding of SQLi mitigation and best practices can be achieved. The primary mitigation strategies that will be examined in this report will be mitigation through the filtering and cleaning of input data as well as the use of prepared statements. Examining each of these mitigation techniques and their strengths can then allow a security professional to implement effective SQLi mitigation. However, before these mitigation techniques can be examined a strong understanding of the SQLi class of vulnerability must be understood and an examination of SQL vulnerabilities will be performed within the context of the Hackazon application. Examining each of these vulnerabilities can then allow for a discussion of principles and best practices for SQLi mitigation within web applications proving a valuable exercise for security professionals to undertake.

SQL injection is one of the most prevalent vulnerabilities found in web applications (OWASP, 2021), and as such it is a strong candidate for security professionals to understand and investigate. SQL injection can be categorized into two primary categories being standard SQL injection and blind SQL injection (OWASP, 2023). The most common and standard form of SQL injection involves the injection of SQL code to perform an action on the underlying database such as the return of unintended data, the modification of data, or the addition and deletion of data (Microsoft, 2023). Blind SQL injection is then performed when more information regarding the underlying structure of the SQL query is needed and can be accomplished through injecting statements that will either always return true or false to observe the database’s behavior (PortSwigger, 2023). The increased understanding gained from blind

SQLi attacks can then allow for more effective injections to accomplish the attacker's goal highlighting the dangers of blind SQLi attacks even though direct data access does not take place. With this in mind, it is clear that SQLi vulnerabilities revolve around the processing of user input, and as such it is essential that the best practices regarding the processing of user input are followed. Whenever user input is accepted that input should be filtered and validated to ensure that only expected input values are accepted into the system (Simpson & Anthill, 2017). As the vulnerabilities found throughout the Hackazon application are examined the filtering of input into a safe and usable form will remain the core principle and methodologies to effectively accomplish this objective will be presented.

The first vulnerability within the Hackazon application can be observed in the product search field (Screenshot 1). This field is vulnerable to SQL injection which is carried out most effectively by prematurely exiting the SQL query with quotation marks and then appending a MySQL comment character to the end of the string to comment out whatever follows (Screenshot 2). When considering how this vulnerability could be mitigated, the first strategy that was attempted was to filter the input so that escape characters were appended to each special character used in the SQL injection attack (Screenshot 3). This was carried out through the use of the addslashes method (PHP, n.d.) but several weaknesses exist in using this method which was soon observed through further examination. The first weakness can be observed in the output of this method where the slash characters can be seen as appended to each special character (Screenshot 4). While this solution works for the purposes of a search field, it would prove problematic for fields where the entry is used in modifying a database entry as the slash characters would be included in the submission (PortSwigger, 2023). Further, while this solution assists in mitigating SQLi attacks it only appends slashes to the entry in the HTML form while the actual processing of the SQL query occurs within underlying JavaScript functions. This highlights the vast importance of understanding the context within which the SQL query is created and processed as the context will determine the mitigation strategies needed to effectively filter malicious input (OWASP, 2023). However, meticulously filtering input as it passes through different contexts is an inefficient solution and, as will be examined next, it is better practice to identify the creation point of the SQL query and to prepare SQL queries in such a way that safe input is always inserted into a pre-crafted query (PHP, n.d.).

While filtering input can work when great care is taken to ensure vulnerabilities have not been missed, it is considered a better practice to prepare SQL queries and then safely pass the user input into that query (PortSwigger, 2023). Within the Hackazon application, this is not as straightforward as the current implementation sees the creation of the SQL query being dynamically generated based on the calling context for the query. However, ensuring safe input at this core area where SQL queries are generated can then allow for the mitigation of SQLi throughout the entire application highlighting an efficient mitigation strategy. For the product search field, this query uses the LIKE keyword to search for items similar to the user input (Screenshot 5). With the current distributed approach to query generation, the input passed to this function could then be cleaned through a method such as mysql_real_escape_string (PHP, n.d.), and the clean input was then passed into the query (Screenshot 6). However, by identifying the functions containing the rest of the SQL query generation (Screenshot 7, Screenshot 9), the query could be prepared with space allocated for the input to be safely inserted into the prepared query (Screenshot 8). Prepared queries are considered the best practice for mitigating SQLi (Microsoft, 2023) and the Hackazon application could be improved by choosing to implement prepared queries in every field where they are used instead of choosing to dynamically generate SQL queries through various function calls.

With this strategy in mind, the remaining vulnerabilities located on the Wishlist (Screenshot 10), review (Screenshot 11), and sign-in (Screenshot 12) forms can be identified and mitigated through centrally preparing SQL queries for safe user input. These vulnerabilities could be mitigated by filtering each point of input entry, but this increases the likelihood of human error and makes maintaining the security of the code more difficult as any changes must be implemented at several locations (Simpson & Anthill, 2017). With these vulnerabilities mitigated, input fuzzing could then be performed again on each field to ensure the vulnerabilities have been mitigated. This can be confirmed by checking the output size of each request and ensuring that the output sizes are roughly equivalent (Screenshot 14). The output size variation is then only representative of the size of the input string as opposed to return data from the database which gave a widely varied initial fuzzing result (Screenshot 13).

SQLi is an attractive vector of attack for malicious actors as the data held within the web application database could be accessed through SQLi attacks (Simpson & Anthill, 2017). Further, SQLi vulnerabilities are one of the most common vulnerabilities found in web applications (OWASP, 2021), and as such it is essential for security professionals to be able to recognize and mitigate SQL injection vulnerabilities. SQL injection can be mitigated through filtering user input as well as preparing SQL queries for safe insertion of user input with prepared statements being preferred as a best practice for increased security (PHP, n.d.). Through examining vulnerabilities in web applications such as Hackazon, a security professional can gain a better understanding of mitigating SQLi within context and in doing so strengthen their ability to secure web applications and mitigate SQL injection vulnerabilities. This then can allow for data and databases to be better protected fulfilling the objectives of security within a web application such as Hackazon.

References

IBM. (January 25, 2023). *Preventing SQL injection attacks*. IBM. https://www.ibm.com/docs/en/db2-for-zos/13?topic=applications-preventing-sql-injection-attacks

Microsoft. (April 3, 2023). *SQL Injection*. Microsoft. https://learn.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver16

OWASP. (2021). *OWASP top ten: Top 10 web application security risks*. OWASP. https://owasp.org/www-project-top-ten/

OWASP. (2023). *SQL injection prevention cheat sheet*. OWASP. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

PHP. (n.d.). *SQL injection*. PHP. https://www.php.net/manual/en/security.database.sql-injection.php#:~:text=The%20recommended%20way%20to%20avoid,provide%20input%20to%20SQL%20statements.

PortSwigger. (2023). *SQL injection*. PortSwigger. https://portswigger.net/web-security/sql-injection

Simpson, M. T., Anthill, N. (2017). *Hands-on ethical hacking (3rd ed.)*. Cengage Learning. https://ng.cengage.com/static/nb/ui/evo/index.html?deploymentId=5681612456081340358553076923&eISBN=9781337271721&id=1937025370&snapshotId=3720027&