ENPM809W Project Phase 3

Dan Fenwick
dfen@umd.edu
Review of ENPM809WProject-hsaglani

Summary

In reviewing the "RestaurantOrderingSystem" web application, I discovered six weaknesses. They vary in severity, but I did not find any that are currently in the CWE "Top 25" list of most dangerous weaknesses. After assessing their severity, I classed the weaknesses as two "Critical" password-related weaknesses, one "High" and one "Low" severity data neutralization weakness, one "Medium" cryptographic weakness and one "Low" cleanup error.

Mitigations for these issues will vary widely in terms of effort and intrusiveness to the existing code base. I recommend prioritizing these issues by developer effort, low to high, rather than by criticality. This will ensure the largest number of issues can be fixed in a given time.

Introduction

I conducted this code review for Harsh Saglani's "RestaurantOrderingSystem" project, on the Phase2 branch, as of the `23ee5f14b1e26a6ac122fab975a006f68526be43` commit, hosted at https://code.umd.edu/hsaglani/ENPM809WProject-hsaglani/-/tree/Phase2.

For the purpose of this review, Harsh asked me to assume the following things:

1. A user does not have access to the administrator computer or the web application server, which are assumed to be physically inside the restaurant and not subject to "evil maid" attack.
2. The system runs on a closed private intranet without access to the public internet. The only devices present on the network are assumed to be the restaurant-provided server and tablets.
3. Sensitive accounts are assumed to have strong passwords, so brute force attacks are supposed to be out of scope.

These assumptions greatly limit the applicability of some of my findings. I'm not entirely sure that such an isolated network is likely to exist in practice, and draconian network security should be a backstop for application-level security, rather than a reason not to secure the application. Therefore, I conclude these weaknesses should be addressed, in order to prevent them from becoming a problem when the deployment environment inevitably changes in the future.

I performed the secure code review as a manual inspection process. Starting from the application home page, I mapped out the reachable pages in the application and visited them from the browser in order to build an approximate map of the "live" source code in the project. I followed this up by reviewing the source code for each page, generally following the philosophy of the "Pull Request" reviewing checklist covered in class.

I attempted to use the static analysis tool PHPStan to augment my manual review, but I found it to be of little use. The tool turned up many flagged items, but from my review, they were not relevant to security. Instead, the tool's findings were about correctness and coding standards. The largest portion were warnings about variables that were possibly uninitialized. The second main category was about undocumented function signatures. While this sort of feedback would be useful in a project with an established coding standard, it was not a fair assessment. The closest thing to a security weakness I found in the list were uses of NULL where a function was documented to require an empty string. However, PHP functions are robust to that sort of minor misuse, and I was unable to find a technical impact for any of PHPStan's findings.

After compiling a list of discovered weaknesses, I made a qualitative assessment of the severity of possible vulnerabilities combined with the likelihood of exploits to assign a criticality score. For example, a SQL injection vulnerability that is only reachable when logged in as an administrator is less dangerous than one that is available to clients before logging in.

Interview

In talks with Harsh Saglani, I determined the following things about his restaurant order system web application.

1. High level functionality: This project is intended to take the place of a restaurant waiter, by serving as a combination menu, order taker, and order fulfillment system.
2. Setup is accomplished by cloning the project source code into an XAMPP installation, which makes the project cross-platform and relatively easy to setup, though not error-proof.
3. Inputs are validated by sanitizing with PHP-bundled filters like filter_var() and mysqli_real_escape_string(), with plans to improve client side input validation in the future.
4. Cryptography is employed chiefly by hashing passwords.
5. Authentication/authorization and session management are performed with built-in PHP functionality, including the bcrypt-based password hash functions and the superglobal $_SESSION object. Additionally, password complexity requirements are enforced when assigning a new password.
6. Error handling, debugging and logging issues do not have security relevance to the code base as it exists at this time.

Code review findings

I located six weaknesses in total, which belong to five CWE weakness identifiers. I grouped my findings by weakness, in ascending numeric order, and then sorted by descending severity.

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Criticality	High
Security category	Data Neutralization Issues
File	implementation/db/db-functions.php
Line: Column	264: 175
Description	The $search_query value is interpolated directly into a query string in order to use the LIKE operator with % wildcards. Although the attacker-controlled string is sanitized with mysql_real_escape_string(), it is not necessarily impossible to exploit: https://stackoverflow.com/questions/5741187/sql-injection-that-gets-around-mysql-real-escape-string
Technical impact	Read Application Data, Bypass Protection Mechanism, Modify Application Data This is hard to exploit, because mysqli_prepare() only allows a single SQL statement. However, no authentication or rate-limiting is applied to

	this query, which means that the prepared statement mechanism and SQL string escaping are the only layers of protection for this weakness.
Mitigations	Parameterize SQL query
Comments	It looks like this approach was intended to work around the difficulty of applying % wildcards with a parameterized query. One possibility is to append the wildcards to the query string before binding the parameter: `$sql_param = "%${search_query}%";`
Criticality	Low
Security category	Data Neutralization Issues
File	implementation/db/admin-db-functions.php
Line: Column	21: 49
Description	The $email value is interpolated directly into a query string. However, the filter_var() function permits dangerous characters that are technically RFC-822 compliant (see https://datatracker.ietf.org/doc/html/rfc822#section-3.3), which allows the inclusion of SQL special characters in the name portion as long as it is wrapped in double quotes, like `"name');#"@b.com`.
Technical impact	Read Application Data, Bypass Protection Mechanism, Modify Application Data This page is restricted to administrator-type users, which reduces the impact. In testing, I was able to get dangerous-looking data through the filter_var() function, but failed to actually affect the SQL query. However, as a matter of principle, all SQL injection is potentially dangerous.
Mitigations	Parameterize SQL query
Comments	N/A

CWE-307: Improper Restriction of Excessive Authentication Attempts

Criticality	Critical
Security category	Authentication Errors
File	implementation/login.php
Line: Column	19: 16
Description	The application does not apply rate limiting to clients or users, allowing an attacker to attempt passwords as fast as the PHP password_verify() function can check a given input.
Technical impact	Bypass Protection Mechanism An attacker could attempt a brute-force attack against a targeted user's password. In addition, because the default bcrypt algorithm used by PHP computationally expensive by design, an attacker making requests at a high rate can cause unexpected CPU load on the application server. Even if all users select passwords that cannot be realistically guessed, this could be used for DDoS attack.
Mitigations	Apply rate limiting, exponential back-off timeouts, or account lockout based on client address or user account.
Comments	N/A

CWE-331: Insufficient Entropy

Criticality	Medium
Security category	Cryptographic Issues
File	implementation/db/admin-db-functions.php
Line: Column	15: 35
Description	A 16-character randomly-generated password ought to have at least 96 bits of entropy, but restricting the output to hexadecimal characters in MD5() reduces that to a maximum of 64 bits. The use of a highly-predictable seed from microtime() further reduces the effective keyspace, which might make the initial account creation phase vulnerable to brute-force attempts, and possibly allow account takeovers. Some entropy is added back in by the str_shuffle() function, but the documentation notes that this is not cryptographic-quality randomness.
Technical impact	Bypass Protection Mechanism An attacker might be able to hijack a brand new account, if they are able to guess the email address and time of creation. The HTTP Date header permits an attacker to accurately estimate the server's time of day, and the email addresses added to the system could be found from other online sources like LinkedIn. Combining this data for targeting purposes could permit account takeover.
Mitigations	Consider a cryptographically-secure function like random_bytes()

Comments	For a 16-character password that that a user can type, try this construct:
	`$random = base64_encode(random_bytes(12));`
	`$password = implode("-", str_split($random, 4));`

CWE-459: Incomplete Cleanup

Criticality	Low
Security category	Initialization and Cleanup Errors
File	implementation/db/db-functions.php
Line: Column	103: 21
Description	One control flow path in the checkout() function leaves the database with autocommit mode turned off. All later database activity on this connection would probably be lost. In order to reach this path, an attacker would have to attempt checkout with invalid food identifiers or quantities; that is, values the database will reject on line 94, resulting in a failed query. These values are partially client-controlled, but subject to validation in the POST handler in the implementation/cart.php file, lines 48-58.
Technical impact	The application uses a long-lived MySQL connection object, created in the implementation/db/db-connect.php file. The PHP-FPM FastCGI server is likely to reuse this object for a long time, so the application could potentially be using a corrupted database object for an extended period. I estimate that the result of this unintentional long-running transaction would be data loss, terminated non-deterministically whenever the system causes an interruption to the database connection.
Mitigations	Restore autocommit mode on the database connection

Comments	Complex control flow with required cleanup is problematic. Important steps could be guaranteed by wrapping the transaction within a function call or using try…catch…finally blocks.

CWE-521: Weak Password Requirements

Criticality	Critical
Security category	Credentials Management Errors
File	implementation/db/validator.php
Line: Column	19: 13
Description	The application applies password complexity requirements, but does not check for the use of easily-guessed dictionary words, like "password" or "iloveyou".
Technical impact	Gain Privileges or Assume Identity Users are very likely to use simple easily-guessed passwords, unless the system requires a more complex password. Because some complexity requirements are enforced, this leads to a false sense of security about the security of user passwords. In conjunction with the lack of rate-limiting on login attempts, an attacker can rapidly search the relatively small keyspace of likely passwords, in order to take over a user account.
Mitigations	Disallow simple dictionary words when setting passwords
Comments	The simplest approach would be to check passwords against a short blocklist like the Twitter banned list. A bigger blocklist taken from SecLists would provide better protection. The best protection would be to integrate complexity estimation from zxcvbn or similar.

Impact

The “RestaurantOrderingSystem” generally avoids disastrous security errors. Most attack vectors are well sanitized, and attacker-controlled inputs are not blindly trusted.

The most likely avenue for compromise of this system is the weak password requirement and lack of login rate limit. An attacker would find this to be the easiest avenue for an intrusion. Depending on the market size when deployed, this could eventually cause significant disruption to the businesses using the system, as well as operational disruption involved with blocking malicious activity and restoring access to rightful users.

The next most likely avenue is a SQL injection vulnerability. I classify this as lower severity due to the difficulty of exploiting the weakness. If an attacker could make it work, it would be even more disruptive to the business than the password weakness, likely resulting in compromise or loss of all data managed by the service.