2013

SQL Injection in SaaS Cloud Layer

Attack and Prevention

To what extent can an SQL injection exploits an SAAS application in the cloud? How vulnerable are today's web-services and are extra precautions taken to increase security when a web-service serves as a portal to a cloud computing service or applications? What are the different kinds of SQL injections possible , what are their patterns , their usage and what solutions can be used to block their attack in a cloud computing platform. This document contains a detailed overview of each of the above mentioned questions.

Clinton D'souza
www.cvdsouza.com
4/30/2013

Clinton D Souza
Practical Solutions to Enhance Security in SaaS Cloud model
Fulton Undergraduate Research Initiative

Final Report
04/30/2013

Practical Solutions to Enhance Security in SaaS Cloud model

Introduction

What is SQL Injection?

Simply put, SQL injection is an insertion of code into a SQL database via a web-service calling said database, where in the injector is able to obtain backdoor access to the database, thus giving him administrative rights or more. [5]

SQL injection is a method of attack where an attacker can exploit vulnerable code and the type of data an application will accept, and can be exploited in any application parameter that influences a database query.
Examples include parameters within the url itself, post data, or cookie values. If successful, SQL Injection can give an attacker access to backend database contents, the ability to remotely execute system commands, or in some circumstances the means to take control of the server hosting the database.[4]

SQL Injection in Multi-Tenant Database Design

We have seen in some of the earlier research about multi-tenant database design that storing the multi-tenant database in the same table segregated by the Tenant ID acting as a primary key is a valid design pattern.
For example if there is a Retail SaaS applications that allows multiple retailers to host their products and sell them through the online SaaS Application, then the possible multi-tenant table design for table that hosts all the orders could be.

SQL Injection types

SQL injection can be categorized into the following types:

A broad classification of injection attacks can be categorized as:

A. First order attack: The attacker can simply enter a malicious string and cause the modified code to be executed immediately.
B. The attacker injects the code into persistent storage (table row) which is deemed as a trusted source. An attack is subsequently executed by other activities by user, server or the database code-itself.
C. Lateral Injection: The attacker can manipulate the implicit function To_Char() by changing the values of the environment variables. NLS_Date_Format or NLS_Numeric_Characters.

SQL injections can be further classified, based on the methodology and vulnerability which makes the attack possible:

1. *Poorly filtered strings.*
    1.1. Caused by user input that is not filtered for escape characters. If validation of the user input if not processed properly, the intruded / malicious user can send SQL strings from a page or URL that takes an input resulting in database input manipulation by the user.

Example :

Attacker puts ‘or”=’ in the form and sends it to the back-end.

SELECT password FROM users WHERE password = OR 1’=’1

This query will return true and thus give he attacker illegal access.

```
$Pass = $_GET[‘pass’];

$password = mysql_query(“SELECT password FROm users WHERE password = ‘”. $pass . “’;”);
```

The query above is a SQL call to SELECT the password from the users database , with the password value being that of $var. IF the user were to input a password that was especially designed to continue the SQL call , it may result in results that were not aforethought. An injection for this may look something like:

‘ OR 1=1

2. *Incorrect type handling.*
    2.1. When an input is not check for type constraints an incorrect type handling occurs.

2.2. An example of this would be an Employee ID what is supposed to be numeric, but there is no filtering in place to check that the user input is numeric.

3. *Signature Evasion.*

3.1. Many SQL injections will be somewhat blocked by intrusion detection and intrusion prevention systems using signature detection rules. Common programs that detect SQL injections are mod_security for Apache and Snort. These programs aren't fool proof and as such , the signatures can be evaded. Some well-known methods that can be used to bypass signature detection are :

3.1.1. Different Encoding: Signature evasion can be made possible with a number of encoding tricks.

> One basic and common encoding trick is the use of URL encoding. URL encoding would change an injection string that would normally look like the following:
>
> NULL OR 1=1
>
> To a URL encoded string that would be masked as:
>
> NULL+OR+1%3D1%2F%2A
>
> Thus the installed IDS system may not register the attack, and the signature will be evaded.

3.1.2. White Space Multiplicity: As common signature databases check for strings such as "OR" (OR followed by a space), it is possible to evade these signatures using different spacing techniques. These techniques can be the use of tabs, new lines/carriage return line feeds, and a variety of other white spaces.

3.1.3. *Arbitrary String Patterns*: In MySQL, comments can be inserted into a query using the C syntax of /* to start the comment, and */ to end the comment. These comment strings can be used to evade signature detection of common words such as UNION, or OR.

4. *Filter Bypassing*: Contains detailed description of techniques that can be used to bypass filters, such as the simple web application.

4.1. *Dollar-Signs*: Although the magic quotes feature has been removed from version 6 of PHP, a lot of web applications still depend on it for security. The magic quotes features calls the addslashes() function on every parameter received in GET and POST requests or in the COOKIE parameter.

5. *Blind SQL injection*.

5.1. Blind SQL Injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page.

SaaS- Multi-Tenant Database: General Design

Multitenancy: The ability to service multiple customers using multiple instances of the same application installed on a single server or multiple server in order to achieve optimal resource utilization. Tenants on a multitenant system can operate as though they have an instance of the software dedicated to them, an instance that is completely secure and insulated from that of other tenants.

Efficiency of a software service is a key element in the survival of a SaaS provider. This efficiency also includes resource utilization, as the price of an application hosted in a cloud environment also includes the amount of memory and data usage needed for the app to run efficiently.

There are three options for Multi-Tenant Database design, these are based on how allocation of databases, space and memory takes place :

1. Separate Database: Each of the tenants is allocated a separate physical database for storing their data.
2. Separate Schema: Each of the tenants is allocated a separate logical unit called schema within a single physical database.

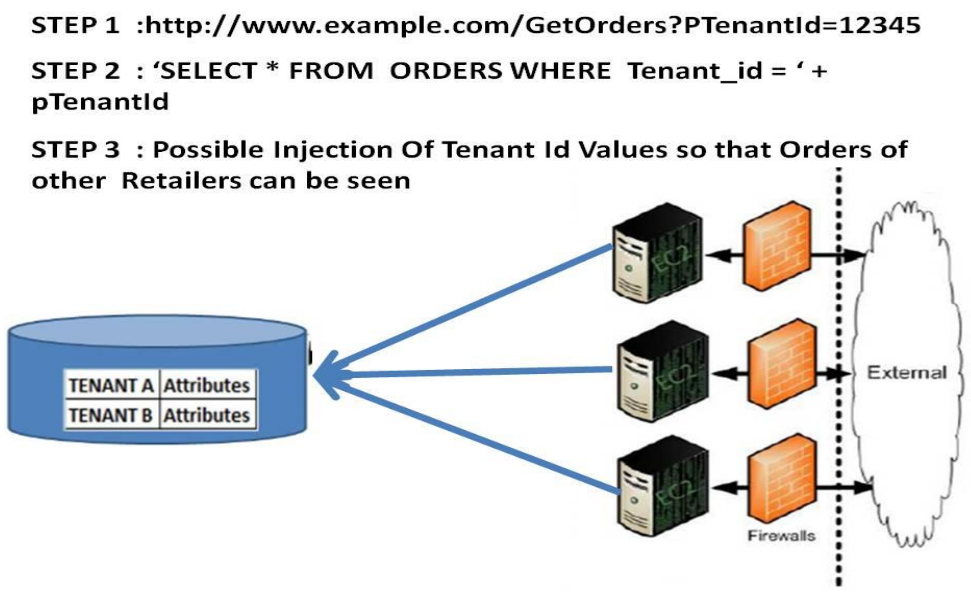
3. Separate Rows: Each of the tenants is allocated the same physical database and schema, but their information is separated using primary keys which re allocated as part of the database design and all the information is kept within the same physical tables.

SQL injection in Multi-Tenant database

Like every database, a multi-tenant database is vulnerable to attacks such as SQL injection. Accessibility to a multi-tenant database from a user perspective is accessed through a web-application, which itself controls the data access.

If the SaaS application is prone to SQL Injection, then it is very easy for some one logging on behalf of One Tenant can view the orders belonging to another Client, let us see with an example.

As explained in this simple example.



STEP 1 :http://www.example.com/GetOrders?PTenantId=12345

STEP 2 : 'SELECT * FROM ORDERS WHERE Tenant_id = ' + pTenantId

STEP 3 : Possible Injection Of Tenant Id Values so that Orders of other Retailers can be seen

- Front end application sends the tenant id to the database calls, so that the orders belonging to that tenant can be retrieved
- However the dynamic SQL written simply attaches the 'tenant Id' Supplied, and returns the results

- This could potentially lead to a situation, where a hacker who could create a dummy 'tenant id' with the SaaS application can pass on random tenant ID values and could view orders belong to other tenants

Best Practices: Problems and Solutions

Although a number of guides providing advice on preventing SQL injections are available, they do not take into consideration the resource utilization and they dynamic nature of muti-tenant databases in a cloud computing environment. The following are advised best practices implementations addressing signature evasion on multi-tenant databases:

- Using a technique called *dynamic profiling web firewall*; the system will automatically establish a profile of minimum and maximum allowable lengths for each URL parameter. The lengths of all incoming parameters are compared to these expected values.
    - Example: When OR 'Simple'='Sim'+'ple' are appended to an existing parameter, the firewall will detect additional characters and will issue a Parameter Length violation.
    - Problem: A single Parameter Length violation is not enough to validate the attack and could either lead to a false positive or false negative.
- To support this profiling web firewall, *a dynamic profiling database firewall* will assist in building a profile of all queries for users of a particular database including web applications accessing the database. This will allow secure scanning of web applications attempting to access a cloud based multi-tenant database.
    - Example: If a web-application passes a new query containing OR 'Simple'='Sim'+'ple' to the back-end database, SecureSphere detects the unusual query and issues an Unknown Query violation.
    - Problem: The single Unknown Query violation is not sufficient to validate an attack. It's possible that a developer may want to add a new query.
- An ideal defense mechanism would be to combine the first two profiling techniques to increase efficiency of the system. This is where a system like SecureSpehre although not perfect comes into action. *A Correlated Attack Validation enforcement*

mechanism which will automatically correlates the Unknown Query with the parameter length violation. These two attack indicators have now been linked resulting in a definitive attack validation. [6]

Conclusion

The best practices proposed, are ideal approaches that would secure a dynamic server and database in the cloud. However, the proposed solution does have its own shortcomings, some of which include inability of the system to deal with blind SQL injection attacks and signature evasion attacks which are usually designed to evade dynamic profiling systems. These attacks usually analyze the signature or heuristics a system uses to analyze incoming parameters and sends queries to bypass these systems.

The proposed solutions will however prevent a malicious web application from sending malicious parameters to a server. Although this is not a fool proof design, the proposed best practices will ensure that a multi-tenant database in a SaaS cloud environment is protected against dynamic attacks via SQL injections and the potential risk of vulnerabilities affecting other users is reduced.

References

1. http://download.oracle.com/oll/tutorials/SQLInjection/html/lesson1/les01_tm_attacks.htm
2. http://www.hakipedia.com/index.php/SQL_Injection

3. http://cloudcomputing.sys-con.com/node/1610582
4. http://iaas.ulitzer.com/node/1624391/mobile
5. http://securecloud.ulitzer.com/node/1624391
6. http://www.issa-sac.org/info_resources/ISSA_20050519_iMperva_SQLInjection.pdf