Example 3

This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. If an attacker with the user name hacker enters the string `"name'); DELETE FROM items; --"` for `itemName`, then the query becomes the following two queries:

```
SELECT * FROM items
WHERE owner = 'hacker'
AND itemname = 'name';

DELETE FROM items;

--'
```

Many database servers, including Microsoft® SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error in Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, in databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (`--`), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed. In this case the comment character serves to remove the trailing single-quote left over from the modified query. In a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1. If an attacker enters the string `"name'); DELETE FROM items; SELECT \* FROM items WHERE 'a'='a"`, the following three valid statements will be created:

```
SELECT * FROM items
WHERE owner = 'hacker'
AND itemname = 'name';

DELETE FROM items;

SELECT * FROM items WHERE 'a'='a';
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from an allow list of safe values or identify and escape a deny list of potentially malicious values. An allow list can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, deny listing is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers can:

- Target fields that are not quoted
- Find ways to bypass the need for certain escaped meta-characters
- Use stored procedures to hide the injected meta-characters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures. Although stored procedures prevent some types of SQL injection attacks, they fail to protect against many others. For example, the following PL/SQL procedure is vulnerable to the same SQL injection attack shown in the first example.

```
procedure get_item (
    itm_cv IN OUT ItmCurTyp,
    usr in varchar2,
    itm in varchar2)
is
    open itm_cv for ' SELECT * FROM items WHERE ' ||
            'owner = '''|| usr ||
            ' AND itemname = ''' || itm || '''';
end get_item;
```

Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.

Related Attacks

- SQL Injection Bypassing WAF
- Blind SQL Injection
- Code Injection
- Double Encoding
- ORM Injection

References

- SQL Injection Knowledge Base - A reference guide for MySQL, MSSQL and Oracle SQL Injection attacks.
- GreenSQL Open Source SQL Injection Filter - An Open Source database firewall used to protect databases from SQL injection attacks.
- An Introduction to SQL Injection Attacks for Oracle Developers
    - This also includes recommended defenses.

Category:Injection