Securing Your Applications Against Injection Attacks

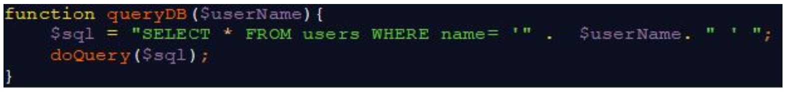
Posted on September 24, 2019 by Ben Dimmick, Jef Fisher

One of the most effective things you can do to protect your company's valuable data is to secure your web applications against injection attacks. The widely respected Open Web Application Security Project (OWASP) publishes a list of the Top 10 Most Critical Web Application Security Risks[1] and injection attacks have topped the list since it was first published in 2010.

SQL Injection Attacks

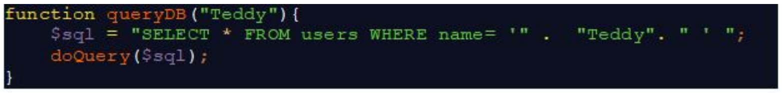
Web applications facilitate interactions between users and server system components, including databases, which are one of the most commonly accessed components. While there are multiple types of injection attacks (e.g., SQL, NoSQL, OS, XML, and LDAP), we will be focusing on attacks that target the interaction between the webserver and the database—otherwise known as SQL injection attacks.

First, we'll review an example of insecure web application code. We'll be using PHP in these examples. However, the concepts here apply to essentially all languages.



```
function queryDB($userName){
    $sql = "SELECT * FROM users WHERE name= '" .  $userName. " ' ";
    doQuery($sql);
}
```

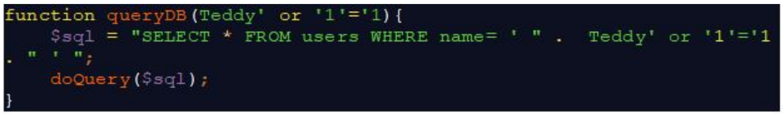
Let's imagine a simple form asking for a user's name. As developers, we expect the user to supply the requested information and submit the form, which causes the user-supplied data to be sent across the web to our application. After entering the application, the data is included in variables that are inserted directly into SQL statements, which are then sent to the database. For example:



```
function queryDB("Teddy"){
    $sql = "SELECT * FROM users WHERE name= '" .  "Teddy". " ' ";
    doQuery($sql);
}
```

As you can see, the code substitutes the user-supplied name ("Teddy") into the SQL statement, which is then sent to the database where the command is interpreted and run. Although the application works as expected in this case, it is NOT secure. It has a vulnerability that can be exploited by entering user-supplied data that consists of some static information and some information that SQL interprets as special characters and code.

For example, what happens if the user supplies content that would change our command to a different one as shown in the next code snippet?



```
function queryDB(Teddy' or '1'='1){
    $sql = "SELECT * FROM users WHERE name= ' " .  Teddy' or '1'='1
. " ' ";
    doQuery($sql);
}
```

As before, the user-supplied data is substituted into the query. However, the code added after the name (or '1'='1) alters the result. The quotes match up to render a valid query,

Recent Tevora Threat Posts

MuleSoft Runtime < 3.8 Unauthenticated RCE (CVE-2019-13116)
October 16, 2019

Smoke and Mirrors | Red Teaming with Physical Penetration Testing and Social Engineering
September 13, 2019

Scout
August 9, 2019

Search for: SEARCH

Recent Posts

Women in Cybersecurity Spotlight: Charlotte Densham, Senior Infosec Associate
October 21, 2019
Charlotte Densham, Senior Information Security Associate at Tevora. …

Women in Cybersecurity Spotlight: Lucy Moreno, InfoSec All-Star
October 21, 2019
Lucy Moreno, Information Security Associate at Tevora. …

Tevora's Cybersecurity Services Achieve ISO/IEC 17020 Accreditation
October 14, 2019
Irvine, CA – October 14, 2019 – Tevora's cybersecurity services …

Categories

Select Category

Archives

Select Month

but instead of only returning the records for which the username is “Teddy” (as in the first example), the query pulls all the records from the database. This is because in all cases, 1 will equal 1.

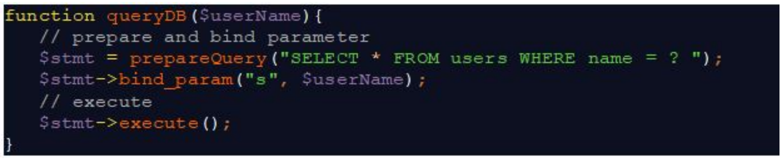
This is one method employed by malicious users to execute SQL injection attacks, which are a huge risk to application security. This method would compromise all rows within a table. Other injection flaws can lead to a complete web server takeover by enabling attackers to execute SQL queries that change existing records.

Defense Against SQL Injection

If you’re constructing SQL statements in your application, make sure you’re using parameterized queries. Not only are parameterized queries invulnerable to SQL injection attacks, but they also have three advantages[2].

1. Prepared statements reduce parsing time because the query is only prepared once.
2. Bound parameters minimize bandwidth to the server as you only need to send the parameters and not the whole query.
3. Prepared statements are a very useful defense against SQL injections because parameter values, which are transmitted later using a different protocol, need not be correctly escaped. If the original statement template is not derived from external input, SQL injection cannot occur.

Here’s an example of a query that is functionally equivalent to the query above, but uses prepared statements that prevent SQL injection:



```
function queryDB($userName){
    // prepare and bind parameter
    $stmt = prepareQuery("SELECT * FROM users WHERE name = ? ");
    $stmt->bind_param("s", $userName);
    // execute
    $stmt->execute();
}
```

As you can see, the username parameter isn’t substituted directly into the SQL statement, but instead is bound to the statement and executed, thereby preventing malicious code.

Another defense is to simply not construct dynamic SQL statements and instead use stored procedures. However, you must be careful how you call the procedures to avoid introducing other vulnerabilities that we’ll cover in future blogs.

References

1. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
2. https://www.w3schools.com/php/php_mysql_prepared_statements.asp

About the Authors

Ben Dimick, Manager, Solutions at Tevora

Jef Fisher, Information Security Analyst at Tevora

Posted in Uncategorized

◂ Tevora Lands on the Orange County Bus…

Women in Cybersecurity Spotlight: Ashli… ▸

Privacy Policy Terms of Service

Data Privacy

Cloud & Security Solutions

Threat Management

Incident Response

Entertainment

Financial Services

Government Entities

Healthcare

Manufacturing & Logistics Industries

Threat Blog

Events & Press

Our Brand

Our Clients