

# Protecting Web Services from Interpretive-Language Injection Attacks

Charles C. Hoffmeyer and Jianhua Wang  
School of Computing and Information Systems  
Grand Valley State University  
Allendale, Michigan  
USA  
cchoffme@cchoffme.com, wangjia@gvsu.edu

## ABSTRACT

The use of trusted input in web based services, specifically services based upon interpretative languages, causes a significant security threat when the input is not properly verified. Malicious input executes undetected with the authority of the system, causing devastating effects. This paper discusses these problems and presents several solutions, then evaluates their effectiveness in an implementation of a password authentication module using ADO .NET and Microsoft SQL Server.

## KEY WORDS

Security Assessment, Design, Interpretive Language, Fault Injection, Input Verification

## 1. Introduction

Security policies and requirements for software have increased significantly over the last few years, due to increased security awareness and the numbers of attacks on web-based systems. “Forty-three percent (43%) of security and law enforcement executives responding report that the total number of... data intrusions increased in 2003 vs. 2002.”[1] Most attacks were the result of exploitation of known ‘bugs’ within the software, relying on an often overworked Systems Administrator to be aware of and apply critical system patches in a timely manner. Often, these patches are either overlooked or introduce more vulnerabilities than they prevent, due to their rush to release.

*“An ounce of prevention is worth a pound of cure.”  
(Proverb)*

This is true of software design and testing. Through the use of a proper testing methodology, given sufficient time, these flaws can be found and fixed prior to the software’s release. Any flaws not found in time are, more often than not, detectable when they are being exploited. Unfortunately this is not true of the interpretive language attack.

Instead of praying on the weakness of a system, attackers will inject the web-based forms with code to gain access to the system. Since these systems rely on interpretation, rather than compiled code, the system cannot distinguish it from the intended data source. Worse yet, there is no way to tell that this intrusion is taking place until after the system has been compromised. This style of attack is often successful due to the naivety of the programmers, database administrators, or the database software itself.

An example of this type of attack can best be seen in SQL-based password authentication modules. In some instances, user input is fed directly to the query. If the user entered a username of “blah” and a password of “foobar”, the following query would be generated and executed:

```
SELECT COUNT(user)
FROM AuthTable
WHERE user = 'blah'
AND password = 'foobar'
```

*Example 1: SQL query that is generated when the user enters a normal username and password in the web service.*

However, a malicious user could type SQL code into the username field (instead of the expected username) and cause the service to generate and execute the following SQL query:

```
SELECT COUNT(user)
FROM AuthTable
WHERE user = "or 1=1 --"
AND password = ''
```

*Example 2: SQL query that is generated when the user enters malicious code into the username field to defeat authentication.*

Because -- is the comment character for SQL, it ignores the requirement to match the password and username. The statement now reads, “To be validated either the username must be null, or 1 must equal 1”. As 1 always equals 1, it authenticates the user.

This paper will present several solutions to this problem and evaluate their merit by implementing them in a .NET-based password authentication module using ADO .NET and Microsoft SQL Server. For each attack pattern discussed in section 2, one or more of the solutions to resolve it will be discussed in section 3. While this paper utilizes SQL when referring to code injection, this style of attack is possible on all systems which may be based partially on any interpretive language.

## 2. Common Interpretive-Language Attack Patterns

Unfortunately there is not just one root cause to the interpretive language injection problem; there are several varying attack patterns which add their own level of difficulty to the problem. Each of these patterns will be discussed in more detail.

### 2.1 Blind Authentication

Blind Authentication is the common method of concatenating user input with an SQL statement to create an executable query to identify the presence of any matching dataset within the database. This is the method used from the example in Section 1.

In this case, the user entered a username and password in a web-based form and pushed the submit button. In doing so, the system looked to see if any rows are returned by the generated query. If the count is anything other than zero, the system permits access.

“This is a relatively common design pattern, and is seen in a number of books and articles.” [2] The problem with this authentication method is that the SQL comment symbol, --, can be used to terminate the query early and still allow a valid authentication by the web service.

There are several ways to handle this problem once it has been identified, but most of the recommended solutions will only reduce the possibility of attack.

### 2.2 User ID Authentication

Two of the possible solutions to the “Blind Authentication” problem in the previous section are to validate the username and to ensure that there was only one row returned by the query (discussed further in section 3). However, this does not completely solve the authentication problem; it only makes it more difficult by requiring that a valid username be known by the intruder. Because a valid username will only return one row when username is key for the table, it will pass these added validity checks.

For a malicious user to perform this intrusion, they would only have to append '-- to the end of a valid username, resulting in the following query for our user “blah”, instead of the expected query seen in example 1:

```
SELECT COUNT(user)
FROM AuthTable
WHERE user = 'blah'-- '
AND password = ''
```

*Example 3: SQL query that is generated when the user enters a valid username followed by a quote and the comment characters.*

Since our user has a valid username, and the number of matching records is only 1, the system will validate this user, even though a password is never entered.

This problem is more difficult to solve because it requires that the organization have a less-than-stringent password security policy, as to not allow a user to have passwords with specific special characters and limit the length of valid passwords.

### 2.3 Data Corruption or Manipulation

Systems may be open to the public, yet still vulnerable to the interpretive language injection attacks. Injection attacks include the possibility of manipulating the data or corrupting it completely. Manipulation attacks are available when the database system executing the commands is running with more permission than is required to perform its task.

Data within the database can be easily manipulated or corrupted when this is the case. A malicious user could add themselves as a valid user to the AuthTable from a simple search field:

```
SELECT *
FROM SearchTable
WHERE keyword like 'mysearch' and
(INSERT INTO AuthTable (user,
password)
VALUES('baduser', 'badpass')) -- '
```

*Example 4: SQL query that is generated from user entering malicious code into the search form on a website. This code causes a valid username and password to be entered into the database.*

While this may cause an error for the search query, the insert would have already been performed, and the user now can log into the protected portion of the system, which is a less than ideal situation. A user with more malicious intents could have done the same query as in example 4, but chose to drop a table instead of inserting the username.

The best way to avoid this problem is to ensure that the database only runs with the least privileges as possible.

## 2.2 Remote Execution of Commands

In [2], a database using Microsoft SQL Server is used to show how an SQL Query can launch executables on the server from remote locations using the following code in the username field of a web service.

```
SELECT COUNT(user)
FROM AuthTable
WHERE user = 'or 1=1 exec
master..xp_cmdshell 'format c: /t'
AND password = ''
```

*Example 5: SQL query that is generated when the user enters a command to remotely format the hard drive in the username field.*

This causes two SQL statements to be executed, one for the original query for valid usernames and one which executes the “format c:” command on the server.

While this type of injection attack requires specific knowledge of the system, it is a concern which should be addressed. Fortunately, this too will only run when the database is given super user privileges. The simple solution is running the database with minimal permission.

## 3. Possible Solutions

The following solutions have been proposed to solve one or more of the interpretive language injection problems. While each of these possible solutions will be discussed in isolation, a better solution may result by combining several of these solutions together.

### 3.1 Limited Text Area for Input

“Don’t allow an attacker enough room to send injections.”[2]

By limiting the text area, the attacker will not have enough room to enter the SQL query to perform the attack. While this works for a username or password field, there are still a couple major problems. The first of which is that some injections could occur yet, such as the common ‘ or 1 = 1 --. The second issue is that some fields intentionally accept long input strings, such as an address field in a HR database or the text field in a search engine. As these long fields are necessary, the service will still be vulnerable.

While many sources ([2], [3], [4]) suggest this solution to the code injection problem, it will only deter attackers. This method should not be used independently of other solutions.

## 3.2 Filtering Input for SQL Keywords

Some sources ([5]) have suggested that harmful keywords be filtered out of the input strings prior to being passed into the SQL query. By doing this, you limit the possibility of a table being dropped or another statement from being concatenated with your own. However, [2] indicates that a comprehensive list could never be created to cover all possible keywords and would only create a false sense of security to the organization. If such a list could be created, there would be great difficulty ensuring that it was maintained throughout the lifecycle of the application.

## 3.3 Input Verification

User input being entered into a web-based service for authentication may be better served with a two-phase authentication method. The first step is our current method of verifying a user as existing in the system, as seen in example query 1, which may be vulnerable to the SQL injection problem. If this query returns as a valid user, the query should be run again to select the username and password for the authenticated user from the database, to be compared with the user’s input.

This string comparison method is performed to ensure what the user entered to authenticate into the system was the same information that was in the database, and did not use injected code:

```
private Boolean checkUser(String userin, String passin){
    ResultSet r = conn.executeQuery(
        "select user, password "+
        "from AuthTable"+
        "where user = '+userin+' and"+
        "password = '+passin+'");
    if ((userin.equals(r.getString(1))){
        If((passin.equals(r.getString(2))){
            return true;
        }
    }
    else{
        return false;
    }
}
```

*Example 6: Java method to authenticate user by method in Example 1 and comparing the database contents string-wise to what was submitted to the database by the web service.*

If a user had entered malicious code into the database, this method would have detected that the code entered into the username or password fields of the web service did not match the username and password that is in the database, and deny access.

### 3.4 Character Replacement

[2], [3], and [5] highly recommend implementing character replacement on the special characters which could be used within an injection attack. By replacing these characters with benign ones, it allows us to no longer be concerned with the standard SQL escape strings which permit these attacks to occur.

However, when using this solution, one must be careful to consider every accepted character set. As [6] indicated, the hexadecimal number 0x27 can be used in place of the ASCII quote character in most SQL databases. With careful planning, character replacement can be highly effective in stopping injection attacks.

### 3.5 Instruction Set Randomization

In [7], Kc, Keromytis, & Prevelakis present an entirely different solution to the entire code injection problem. Their method is to create “process-specific” randomized instruction sets at the processor level. In order for an attacker to launch injected code, they would need to know the secret randomization key.

While this method is ideal in preventing injection attacks, it requires processor support (which currently is only the Transmeta Crusoe) and both operating system and database server software be significantly modified. Because of these requirements, it is mostly a theoretical solution at this point. However, we suspect that as the injection problem worsens, we will see x86 processors and others supporting this functionality.

## 4. Testing Solutions

To test the proposed solutions, we have written a password authentication module using ADO .NET, using a Microsoft SQL Server database to support our queries.

Each of the solutions discussed in sections 3.1 to 3.4 are implemented and discussed below as to their effectiveness in stopping the common attacks presented. In addition, a combination of the solutions presented will be evaluated to see if the combined effect is greater than the sum of its parts.

In implementing this solution, we were quite surprised to find that SQL Server does not allow the escape characters to be used in a query, where in other databases (MySQL, McKoi) the escape characters could be freely used. Because of this we had to use the whole query, as intended by the database administrator, in the injection attack.

We expected that the order of execution for the query would be from left to right, but that is not what we found.

The “AND” clause took precedence, followed by the “OR”. In using the injected code, as seen in Example 7, this ordering requires that we know at least one piece of information requested by the system. For this example, it is either a valid username or a valid password.

```
SELECT *
FROM AuthTable
WHERE user = 'RobertB' OR '1'='1'
AND password = ''
```

Example 7: Injection query used in for the test application.

### 4.1 Results from the “Limited Text” Solution

The “Limited Text” solution is used to truncate any attempt to inject code into the web applet. In Image 1, we have limited the username field to 10 characters.

This solution was able to stop **RobertB' OR '1'='1** from being used in the username field. However, if the field size was set to 12 instead of 10, a short username could be used and the injection attack would be successful.

#### SQL Injection Demo

Your login name and password are not correct.  
Please try again.

Image 1: “Limited Text” solution.

### 4.2 Results from the “Filtering Keywords” Solution

Keywords are filtered out to stop users from being able to manipulate the data within our database through the interface fields. In doing this, manipulation attacks cannot be performed. As mentioned before, continued success relies on a consistently updated keyword list in every character map accepted by the database. Image 2

shows the result after removing the keywords from the input string.

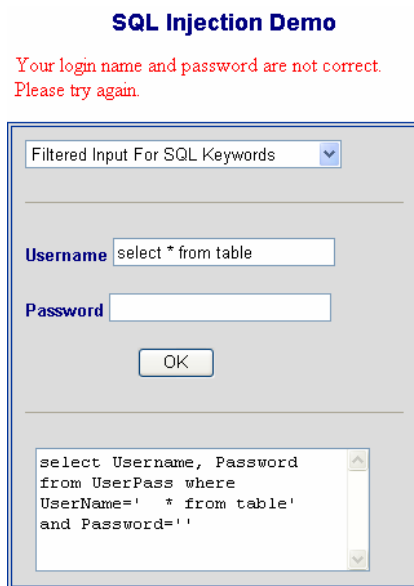


Image 2: Input filtered for main SQL keywords.

### 4.3 Results from the “Input Verification” Solution

Input Verification is the two-phase approach: it first confirms that the database is returning the user as authorized user, and then it confirms that the data being used as input to the query string is the same data that is in the database.

This solution was able to stop **RobertB’ OR ‘1’=’1** from being successful in gaining entry to the system, where it may not have been stopped by either of the previously mentioned methods.

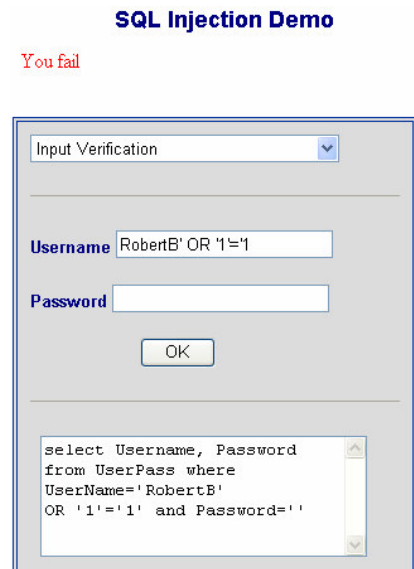


Image 3: Input Verification to confirm valid data input. .

### 4.4 Results from “Character Replacement”

For this case, we removed any character that could be used as an escape character, as well as all quotes, parenthesis, comparator symbols and asterisks. This was done for each character map available to the database. This ensures that an injection attack cannot occur, due to the key characters not being available to the user.

As previously mentioned in the solution discussion, the caveat to this problem is that it must be done for all character maps or it will not be effective.

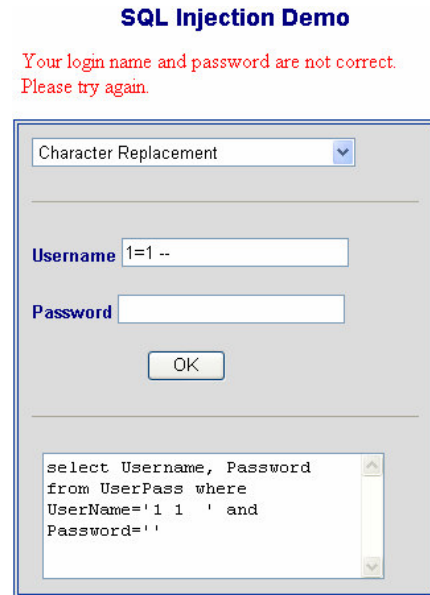


Image 4: Character Replacement

### 4.5 Results from Combining Solutions

In combining these solutions together, we could not force the system to allow us entry through the use of injected code.

## 5. Conclusion

Several problems have been identified in the use of interpretive languages in web-based services, along with many possible solutions. While each of these solutions has its own weaknesses, strength comes from unity. By combining these solutions, a user cannot inject code through the web service.

From Table 1, below, it is evident that there are two solutions which can be used to stop interpretive language injection attacks from being successful: Limiting text area for user input, and Character Replacement.

	3.1	3.2	3.3	3.4
Blind Authentication	X		X	X
User ID Authentication	X		X	X
Corruption / Manipulation	X	X		X
Remote Execution	X	X		X

Table 1: Summary of Problems and Possible Solutions

As we saw from our example program, simply limiting the text area does not solve the problem completely. It should not be used in isolation. However, by including strong character replacement in conjunction with limiting the text area (when possible), interpretive language injection attacks can be prevented.

In the case of user authentication, or any case where the user is confirming information with the database, input verification is an ideal solution. Not only does it ensure that the information entered into the database was not injected code, but it is a means to confirm that the data going into the query matches the data in the database.

## References:

- [1] CERT® Coordination Center, *2004 E-Crime Watch Survey: Summary of Results*, [Internet] <http://www.cert.org/archive/pdf/2004eCrimeWatchSummary.pdf>
- [2] Breidenbach, *Guarding your website against SQL injection attack* (Berkley, CA: APress, 2003).
- [3] Viega & McGraw, *Building secure software: how to avoid security problems the right way* (Boston, MA: Addison-Wesley Professional Computing Series, 2004).
- [4] Y. Huang, S. Huang, T. Lin, & C. Tsai. Web application security assessment by fault injection and behavior monitoring, *WWW 2003: Proceedings of the twelfth international conference on World Wide Web*, Budapest, Hungary, 2003, 148-159.
- [5] Spriet, Real-world PHP security, *Linux Journal*, 2004(120), 2004.
- [6] Howard & Leblanc, *Writing secure code: 2<sup>nd</sup> edition* (Redmond, WA: Microsoft Press, 2003).
- [7] Kc, Keromytis, & Prevelakis. Countering code-injection attacks with instruction-set randomization, *Proceedings of the 10<sup>th</sup> ACM conference on Computer and Communications Security*, New York, New York, 2003, 272-280.