

# Malicious Code Injection: It's Not Just for SQL Anymore

By Bryan Sullivan

More and more, developers are becoming aware of the threats posed by malicious code, and SQL injection in particular, and by leaving code vulnerable to such attacks. However, while SQL is the most popular type of code injection attack, there are several others that can be just as dangerous to your applications and your data, including LDAP injection and XPath injection. While these may not be as well-known to developers, they are already in the hands of hackers, and they should be of concern.

In addition, much of the common wisdom concerning remediation of malicious code injection attacks is inadequate or inaccurate. Following these flawed recommendations will not improve the security of your application, but will only leave you with a false sense of security until the next time your application is compromised and your data is stolen, erased, or tampered with. It is important for developers to acquaint themselves with all code injection types that exist as well as the proper ways to fix any vulnerabilities to malicious code.

## The Basic Premise of All Code Injection Types

Many people mistakenly think that they are safe from malicious code injection attacks because they have firewalls or SSL encryption. However, while a firewall can protect you from network level attacks, and while SSL encryption can protect you from an outside user intercepting data between two points, neither of these options offers any real protection from code injection attacks. All code injection attacks work on the same principle: a hacker piggybacks malicious code onto good code through an input field in the application. Therefore, the protection instead has to come from the code within the application itself.

There are many motives that hackers using malicious code injection attacks may have. They may wish to access a website or database that was intended only for a certain set of users. They may also wish to access a database in order to steal such sensitive information as social security numbers and credit cards. Other hackers may wish to tamper with a database – lowering prices, for example, so they can steal items from an e-commerce site with ease. And once an attacker has gained access to a database by using malicious code, he may even be able to delete it completely, causing chaos for the business that has been attacked.

The root of all code injection problems is that developers put too much trust into the users of applications. A developer should never trust the user to operate the application in a safe manner. There will always be someone who is looking to use malicious code in an exploitative manner.

## Looking Beyond SQL Injections

Aside from SQL injections, there are several other types of malicious code injection attacks with which developers must become familiar. Three of these types of dangerous malicious code injections are XPath injection, LDAP injection, and command execution injection.

An XPath injection attack is similar to an SQL injection attack, but its target is an XML document rather than an SQL database. The attacker inputs a string of malicious code meant to trick the application into providing access to protected information. If your website uses an XML (Extensible Markup Language) document to store data and user input is included in an XPath query against that document, you may be vulnerable to an XPath injection.

For example, consider the following XML document used by an e-commerce website to store customers' order history:

```
<?xml version="1.0" encoding="utf-8" ?>
<orders>
  <customer id="1">
    <name>Bob Smith</name>
    <email>bob.smith@bobsmithinc.com</email>
    <creditcard>1234567812345678</creditcard>
    <order>
      <item>
        <quantity>1</quantity>
        <price>10.00</price>
        <name>Sprocket</name>
      </item>
      <item>
        <quantity>2</quantity>
        <price>9.00</price>
        <name>Cog</name>
      </item>
    </order>
  </customer>
  ...
</orders>
```

The website allows its users to search for items in their order history based on price. The XPath query that the application performs looks like this:

```
string query = "/orders/customer[@id='" + customerId +
"']/order/item[price >= '" + priceFilter + "']";
```

If both the customerId and priceFilter values have not been properly validated, an attacker will be able to exploit the XPath injection vulnerability. Entering the following value for either value will select the entire XML document and return it to the attacker:

```
' ] | /* | /foo[bar='
```

With one simple request, the attacker has stolen personal data including e-mail addresses and credit card numbers for every customer that has ever used the website. Blind XPath injection attacks, like blind SQL injection attacks, are possible, but in situations like our example they're not even necessary. XPath queries do not throw errors when the search elements are missing from the XML document in the same way that SQL queries do when the search table or columns are missing from the SQL database. Because of the forgiving nature of XPath, it can actually be easier for an attacker to use malicious code to perform an XPath injection attack than an SQL

injection attack.

### **LDAP Injection and Command Execution**

Like SQL injection for SQL databases and XPath injection for XML documents, LDAP injection attacks provide the malicious user with access to an LDAP directory, through which he or she can extract information that would normally be hidden from view. For example, an attacker could possibly uncover personal or password-protected information about a professor listed in the directory of a collegiate site. A hacker using this technique may rely on monitoring the absence and presence of error messages returned from the malicious code injection to further pursue an attack.

Some examples of LDAP injection clauses are:

- \*
- )|(cn=\*)
- )|(objectclass=\*)
- )|(homedirectory=\*)

Finally, command execution can also provide the means for malicious code injection. Many times, a website calls out to another program on the system to accomplish some kind of goal. For example, in a UNIX system, the finger command can be used to find out details about when a user was last on the system, for how long, and so on. A user could, in this case, attach malicious code to the finger command and gain access to the system and its data process. So, the command:

finger **bobsmith**

becomes:

finger **bobsmith; rm -rf /**

which will attempt to delete every file on the system.

### **Preventative Measures: The Good and the Bad**

Several preventative actions have commonly been suggested to developers to protect applications from malicious code injection, but many of these have proven inadequate. For example, developers are told that turning off error messages can prevent code injection attacks, which is untrue. Some code injection attacks do not rely on error messages at all. These attacks are called “blind” injections. Since blind injection attacks can succeed even if error messages are suppressed, turning off error messages simply makes the application more obscure for the legitimate user while leaving data vulnerable to attack.

In addition, it is often said that using stored procedures for SQL calls can help remove vulnerabilities to SQL injections. This approach is easy to take with many applications – Oracle databases allow the user to write stored procedures in Java, while Microsoft SQL Server 2005 allows stored procedures to be written in .NET languages like C#. While there are many good reasons to use stored procedures, they do not solve the problem of SQL injection on their own. Using stored procedures simply shifts the burden of the

problem onto the stored procedures. The complicated languages that allow the writing of stored procedures also are open to programming mistakes – mistakes that can lead to code injection vulnerability. The bottom line is that the developer has the responsibility to ensure that the data that is being passed to a database is safe and secure, so more steps must be taken at this stage.

The only real way to defend against all malicious code injection attacks is to validate every input from every user. While establishing a list of “bad” input values that should be blocked (a blacklist) may seem like an appropriate first step, this approach is extremely limited. A finite list of problems simply gives hackers the opportunity to discover ways around your list. There is simply no way to make sure that you are covering every possibility with your blacklist, so you are still leaving the application vulnerable to malicious code injections.

The correct way to validate input is to start instead with a whitelist – a list of allowable options. For example, a whitelist may allow usernames that fit within specific parameters – only eight characters long with no punctuation or symbols, and so on. This can reduce the surface area of a malicious code injection attack by specifying the proper format for the input into the field. The application can then reject input that does not fit the established format. This approach (unlike a blacklist) can prevent not only known, current attacks but also unknown, future attacks.

To be completely thorough, a developer should set up both white- and blacklists in order to cover all bases. In this way, the whitelist can be used to block the majority of attacks, while the blacklist can cover specific edge cases not handled by the whitelist. To protect against SQL injection, a whitelist could allow only alphanumeric input, while a “backup” blacklist could specifically disallow common SQL verbs like SELECT and UPDATE.

## **Conclusion**

Developers may already be aware of SQL injections, but they may not be considering other types of malicious code injection attacks when creating a web application. Many applications are therefore left vulnerable to attack. A good developer should familiarize him or herself with other types of code injection, including LDAP injection and XPath injection, as well as the best ways to stop these attacks. In this way, applications can be made more secure at the start of the development process, and data will be protected.

## **About the Author**

Bryan Sullivan is a development manager at SPI Dynamics, a [Web application security products company](http://www.spidynamics.com/) (<http://www.spidynamics.com/>) Bryan manages the DevInspect and QAInspect Web security products, which help programmers maintain [application security](#) throughout the development and testing process. He has a bachelor’s degree in mathematics from Georgia Tech and 11 years of experience in the information technology industry. Bryan is currently coauthoring a book on Ajax security, which will be published in summer 2007.