View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CiteSeerX

Eliminate Sql Injection Using LINQ

[I]V. Vilasini, [II]P. Chellamal

[I]ME II CSE, JJ College of Engineering and Technology, Thiruchirapalli, India

[II]Assistant Professor, JJ College of Engineering and Technology, Thiruchirapalli, India

Abstract

As web application security breaches and attempts rise, developers are increasingly being asked to take more responsibility for the security of their applications. In fact security-related concerns are hard to apply as they involve adding complexity to already complex code. In this paper, we have proposed a lightweight approach to prevent SQL Injection attacks, that it can actually be well defended by using LINQ (Language Integrated Query) .LINQ to SQL, when used exclusively for data access, eliminates the possibility of SQL injection in your application for one simple reason: every SQL query that LINQ executes on your behalf is parameterized. Internally, it means that when LINQ to SQL queries the database, instead of using plain values, it passes them as SQL parameters, which means they can never be treated as executable code by the database. This is also true for most (if not all) ORM mappers out there.

Keywords

SQL Injection Attacks , LINQ, Web security, Injection prevention, OWASP

I. Introduction

Many developers have learned that the most effective way to build secure applications and prevent damaging attacks is to design and implement the applications securely from the beginning. Unfortunately, development teams often lack the training and resources to make educated design decisions about application security. As developers assume more of the security burden, the first web application vulnerability that many developers learn about is a particularly dangerous form of command injection known as SQL injection. Command injection in its archetypal form is any vulnerability that allows an attacker to run an unintended command on your server by providing unanticipated input that alters the way you intended the web application to run. Because it's so well-known, SQL injection attacks are common, dangerous, and pervasive. Fortunately, you can prevent SQL injection easily once you understand the problem. Even better, a new Microsoft data access technology offers .NET developers the opportunity to eliminate SQL injection vulnerabilities altogether—when used properly. That technology is called Language Integrated Query (LINQ). This paper explores LINQ's potential for hardening your web application's data access code so that it's impossible to attack through SQL Injection.

II. Overview

SQL injection is a type of web application security vulnerability whereby an attacker supplies malicious data to the application, tricking it into executing unanticipated SQL commands on the server. These attacks are fairly easy to prevent, but they're also both common and pernicious because they allow attackers to run database commands directly against your production data. In the most extreme cases, attackers can not only gain unfettered access to all of your data, but can also drop tables and databases or even gain control of the database server itself. If these attacks are easy to prevent, then why are they so dangerous? First, your application database is a very attractive target for obvious reasons and garners a lot of attention from attackers. When SQL injection is possible in a web application, it is very easy for an attacker to detect it and to then exploit it. So it stands to reason that even if SQL injection mistakes are not the most frequent security mistakes made by developers, they very well may be the most frequently uncovered and exploited in the wild.

One easy way to detect SQL injection vulnerability is to insert a meta character into an input that you know an application will use to craft a database access statement. For example, on any web site that contains a search input field, an attacker can input a database meta character such as a tick mark (') and click the Search button to submit the input. If the application returns a database error message, the attacker not only knows that he has found a database-driven portion of the application, but also that he may be able to inject more meaningful commands and have your server execute them. Application security researcher Michael Sutton recently emphasized the ease of discovering web applications vulnerable to SQL injection by identifying hundreds of potentially vulnerable sites in a matter of minutes using the Google search API .

A. Anatomy of SQL Injection

Here's a simple SQL injection example walkthrough to demonstrate both how easy the mistakes are to make and how simple they can be to prevent with some design and programming rigor. The sample web application contains a simple customer data search page named SQLInjection.aspx that is vulnerable to SQL injection. The page contains a Company Name input server control and a data grid control to display the search results from the Microsoft sample Northwind database that ships with SQL Server 2005 Express Edition. The query executed during the search includes a very common mistake in application design—it dynamically builds a SQL query from user-provided input. This is the cardinal sin of web application data access because it implicitly trusts what the user posts, and sends it straight to your database. The query looks like this when initiated from the Search button click event:

```
  protected void btnSearch_Click(object sender, EventArgs e)
  {
      String cmd = "SELECT [CustomerID], [CompanyName],
[ContactName]
          FROM [Customers] WHERE CompanyName ='" +
txtCompanyName.Text
       + "'";

     SqlDataSource1.SelectCommand = cmd;
     GridView1.Visible = true;
  }
```

In the intended scenario, if a user inputs "Ernst Handel" as the company and clicks the Search button, the response shows the customer record for that company, as expected. But an attacker could easily manipulate this dynamic query, for example, by inserting a UNION clause and terminating the rest of the intended statement with comment marks (—). In other words, instead of entering “Ernst Handel,” the attacker would input the following:

```
Ernst Handel’ UNION SELECT CustomerID, ShipName,
ShipAddress
FROM ORDERS--
```

The result is that the SQL statement executed on the server ends up appending the malicious request. It transforms the dynamic SQL to this:

```
SELECT [CustomerID], [CompanyName],
   [ContactName]   FROM [Customers]
 WHERE CompanyName =’Ernst Handel’
 UNION SELECT CustomerID, ShipName,
   ShipAddress   FROM ORDERS--’
```

This is a perfectly legal SQL statement that will execute on the application database, returning all the customers in the Orders table who have processed orders through the application.

B. Kinds of SQL injection

1. In-band
Also called Error-based or Union based SQL Injection or first order Injection. Here communication between the attacker and the application happens through a single channel.

2. Out-band
This kind of an attack uses two different channels for communication between attacker and the application.

3. Inferred
Also known as Blind – SQL – Injection. Here the server doesn’t respond with any syntax error or other means of notification.The attacker needs to retrieve the data by asking true or false questions through SQL commands.

C. Typical SQL Safeguards
You can see now how easy it is to both create a SQL injection vulnerability in your application and to exploit it. Fortunately, as mentioned before, SQL injection can usually be prevented easily with a few simple countermeasures. The most common and cost effective way to prevent SQL injection is to properly validate all inputs in the application that are ultimately used as data access. Any input that originates with users—either directly through the web application or persisted in a data store—must be validated on the server for type, length, format and range before processing your data-access commands on the server. Unfortunately, code-based countermeasures are not foolproof and can fail when:

- Validation routines aren’t properly designed.
- Validation is performed only on the client layer.
- Validation misses even a single field in the application.

An additional layer of defense to prevent SQL injections involves properly parameterizing all the SQL queries in your application, whether in dynamic SQL statements or stored procedures. For example, the code would have been safe if it had structured the query like the following:

```
SELECT [CustomerID], [CompanyName], [ContactName]
FROM [Customers]
WHERE CompanyName = @CompanyName
```

Parameterized queries treat input as a literal value when executed as part of the SQL statement; thereby making it impossible for the server to treat parameterized input as executable code. Even if you use stored procedures, you must still take this extra step to parameterize input, because stored procedures provide no inherent protection from SQL injection over embedded queries. Even with these simple fixes, SQL injection is still a big problem for many organizations. The challenge in your development team is to educate every developer about these types of vulnerabilities, put meaningful and effective security standards in place to prevent attacks, enforce the standards and conduct security assessments to validate that nothing was missed. This introduces a lot of variables in your efforts to secure your applications, so you would be much more productive if you were to select a data-access technology that renders these SQL injection attacks impossible. This is where LINQ comes in.

III. LINQ Overview
At its simplest, LINQ adds standard patterns for querying and updating data in any type of data store—from SQL databases to XML documents to .NET objects. When building database-driven applications, the component of LINQ that enables developers to manage relational data as objects in C# or VB is known as “LINQ to SQL,” which is considered part of the ADO.NET family of data technologies. When originally introduced in CTP form, LINQ to SQL was known as DLINQ.

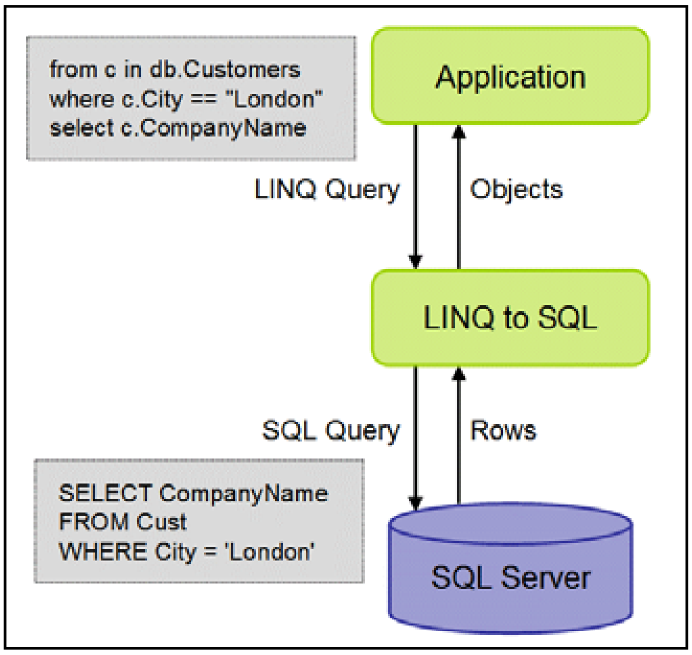

Fig. 1: Execution Architecture

LINQ to SQL enables you to treat data in your applications as native objects in the programming language you are using, abstracting the complexity of relational data management and database connections. In fact, you can display and manipulate database data through LINQ without writing a single SQL statement. At runtime, LINQ to SQL translates queries embedded or “integrated” in your code into SQL, and executes them on the database. LINQ to SQL returns the query results to the application as objects, completely abstracting your interaction with the database and SQL. There is no faster way to eliminate the possibilities of SQL injection in web applications than to eliminate SQL from your application. With LINQ to SQL, you can do that.

IV. Securing Data Access with LINQ

LINQ to SQL, when used exclusively for data access, eliminates the possibility of SQL injection in your application for one simple reason: every SQL query that LINQ executes on your behalf is parameterized. Any input provided to the query from any source is treated as a literal when LINQ builds the SQL query from your embedded query syntax. Furthermore, LINQ's integration with Visual Studio Orcas assists developers in building valid queries through IntelliSense and compile-time syntax checking. The compiler catches a lot of query misuse that might introduce functional defects or other types of vulnerabilities into your application. In contrast, SQL statements you write are parsed and interpreted on the database only at runtime before you know whether it is correct or not. The only attack vector against LINQ to SQL is for an attacker to try to trick LINQ into forming illegal or unintended SQL. Fortunately, the languages and compilers are designed to protect you from that.

With that in mind, here's how you can implement the customer search example using LINQ to SQL to protect against SQL injection attacks. The first step is to create the object model of the relational data in the database. Visual Studio Orcas includes a new Object Relational Designer (O/R Designer) that enables you to generate the full object model for your database by dragging tables onto the design surface and defining relationships. To build the object model for our Northwind Customers table, you create a LINQ to SQL database file in your application by selecting "Add New Item…" on your project and choosing the "LINQ to SQL File" template, which opens in the O/R Designer. To automatically build the complete object model for the Customers table, select that table in the Server Explorer and drag it on to the O/R Designer design surface. In this example, the O/R Designer adds a file named Customers.designer.cs that defines the classes you'll use in code rather than writing code to interact directly with the database.

After defining the object model classes for the data in the Customers table, you can query the data directly in code for the customer data search page. The Page_Load method for the LINQ-powered page (LINQtoSQL.aspx.cs), instantiates the CustomersDataContext class created by the O/R Designer, reusing the same connection string used previously in the SQLInjection.aspx page. The LINQ query below retrieves a collection of Customer objects that match my where clause:

```
protected void Page_Load(object sender, EventArgs e)
  {
    string connectionString =
      ConfigurationManager.ConnectionStrings
      ["northwndConnectionString1"].ConnectionString;

    CustomersDataContext db = new
      CustomersDataContext(connectionString);

    GridView1.DataSource =
        from customer in db.Customers
        where customer.CompanyName ==
        txtCompanyName.Text
        orderby customer.CompanyName
        select customer;
    GridView1.DataBind();
  }
```

Using LINQ to SQL, if I provide "Ernst Handel" as the Search value, the SQL statement generated by LINQ at runtime and executed on the server looks like this:

```
 SELECT [t0].[CustomerID], [t0].[CompanyName],
[t0].[ContactName],[t0].[ContactTitle],[t0].[Address],
   [t0].[City], [t0].[Region], [t0].[PostalCode], [t0].[Country],
   [t0].[Phone], [t0].[Fax]
 FROM [dbo].[Customers] AS [t0]
 WHERE [t0].[CompanyName] = @p0
 ORDER BY [t0].[CompanyName]}
```

As you can see, the WHERE clause is parameterized automatically; therefore, it's impervious to attack with conventional SQL injection attacks. No matter what values a user provides as input to the search page, this query is type-safe and will not allow input to execute commands on the server. If you input the attack string used earlier for the SQL injection exploit, the query returns no rows. In fact, the most harm that a user could do with this query is to perform a brute force attack, using the search function to enumerate all the company records in the Customers table by guessing every possible value. But even that only provides the Customers values already exposed on that page, and gives attackers no opportunity to inject commands that provide access to additional tables or data in the database.

V. Conclusion

As the examples have shown, it's easy to introduce SQL injection vulnerabilities into web applications, and easy to fix them with proper diligence. But nothing inherently protects developers from making these simple, yet dangerous mistakes. However, Microsoft's LINQ to SQL technology removes the possibility of SQL injection attacks from database applications by letting developers interact directly with object models generated from relational data rather than directly with the database itself. The LINQ infrastructure built into C# and Visual Basic takes care of formulating legal and safe SQL statements, preventing SQL injection attacks and enabling developers to focus on the programming language most natural to them.

References

[1] *Atefeh Tajpour and Maslin Massrum, "Comparison of SQL Injection Detection and Prevention Techniques," in 201O 2nd International Conference on Education Technology and Computer (ICETC).*

[2] *Debabrata Kar and Suvasini Panigrahi, "Prevention of SQL Injection Attack Using Query Transformation and Hashing"in 2013 3rd IEEE International Advance Computing Conference (IACC).*

[3] *Diallo Abdoulaye Kindy and Al-Sakib Khan Pathan, "A Survey On Sql Injection: Vulnerabilities, Attacks, And Prevention Techniques" in 2011 IEEE 15th International Symposium on Consumer Electronics.*

[4] *David Byers, Nahid Shahmehri, "Unified modeling of attacks, vulnerabilities and security activities," Proc. 2010 ICSE Workshop on Software Engineering for Secure Systems, IEEE, 2010.*

[5] *Elizabeth Fong, Romain Gaucher, Vadim Okun, Paul E.Black, Eric Dalci, "Building a test suite for web application scanners," Proc. Annual Hawaii International Conference on System Sciences, IEEE.*

[6] *Ezumalai.R and Aghila.G," Combinatorial Approach for Preventing SQL Injection Attacks" in 2009 IEEE International Advance Computing Conference (IACC 2009).*

[7] *Jason Bau, Elie Bursztein,Divij Gupta,John Mitchel, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," 2010 IEEE Symposium on Security and Privacy. IEEE, 2010.*

[8] *Jie Wang, Raphael C.-W. Phan, John N. Whitley, David J. Parish, "Augmented Attack Tree Modeling of SQL Injection Attacks," Proc. 2nd IEEE International Conference on Information Management and Engineering, IEEE, 2010.*

[9] *Ke Wei, M. Muthuprasanna and Suraj Kothari. Preventing SQL Injection Attacks in Stored Procedures. IEEE, 2006.*

[10] *Lwin Khin Shar and Hee Beng Kuan Tan, "Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities," in ICSE 2012.*

[11] *MeiJunjin, " An approach for SQL Injection vulnerability detection" IEEE,2009.*

[12] *Nuno Antunes and Marco Vieira, " Detecting SQL Injection vulnerabilities in web services" IEEE,2009.*

[13] *TIAN Wei, YANG Ju-Feng and XU Jing, SI Guan-Nan, "Attack model based penetration test for SQL injection vulnerability," 2012 IEEE 36th International Conference on Computer Software and Applications Workshops.*

[14] *Robert Dollinger and Kent Thomas,"Using LINQ Transformation Patterns to evaluate SQL Queries"in 2011 IEEE.*