작성자 : 기술지원부 조 태 준 tedcho@nextline.net

MS-SQL Database 백업하기

ASP.NET 및 Microsoft SQL Server과 같은 강력한 데이터베이스 서버의 고급 서버측 기술을 통해 개발자는 동적인 데이터 중심 웹 사이트를 매우 쉽게 만들 수 있습니다. 하지만 ASP.NET 및 SQL의 기능은 SQL injections 공격이라는 너무나 일반적인 공격 방식을 알고 있는 해커들에게도 쉽게 악용될 수 있습니다.

SQL injections 공격에 대한 기본 개념은 다음과 같습니다. 사용자가 텍스트 상자에 텍스트를 입력할 수 있도록 웹 페이지를 만들고 이러한 텍스트는 데이터베이스에 대한 쿼리를 수행하는데 사용됩니다. 해커는 이러한 텍스트 상자에 쿼리의 특성을 변경하여 백엔드 데이터베이스에 침입하거나 데이터베이스를 손상시킬 수 있는 잘못 형성된 SQL 문을 입력합니다.

1. SQL 문의 변환

여러 ASP.NET 응용 프로그램에서는 아래와 같이 표시된 것과 같은 폼을 사용하여 사용자를 인증합니다.

```
private void cmdLogin_Click(object sender, System.EventArgs e) {
    string strCnx =
     "server=localhost;database=northwind;uid=sa;pwd=;";
     SqlConnection cnx = new SqlConnection(strCnx);

    cnx.Open();

    //This code is susceptible to SQL injection attacks.
    string strQry = "SELECT Count(*) FROM Users WHERE UserName='" +
     txtUser.Text + "' AND Password='" + txtPassword.Text + "'";
    int intRecs;

    SqlCommand cmd = new SqlCommand(strQry, cnx);
    intRecs = (int) cmd.ExecuteScalar();

    if (intRecs>0) {
        FormsAuthentication.RedirectFromLoginPage(txtUser.Text, false);
    }
    else {
        lblMsg.Text = "Login attempt failed.";
    }

    cnx.Close();
}
```

표 1

사용자가 BadLogin.aspx의 Login 단추를 클릭하면 사용자가 폼의 텍스트 상자 컨트롤에 입력한 값과 UserName 및 Password가 일치하는 Users 테이블에 있는 레코드 수를 계산하는 쿼리를 실행하여 cmdLogin_Click 메서드가 사용자를 인증하도록 시도합니다.

대부분의 경우 폼은 정확히 의도된 대로 작동합니다. 사용자는 Users 테이블에 있는 레코드와 일치하는 사용자 이름 및 암호를 입력합니다. 동적으로 생성된 SQL 쿼리를 사용하여 일치하는 행의 개수를 검색합니다. 그런 다음 사용자를 인증하고 요청된 페이지로 리디렉션합니다. 잘못된 사용자 이름 및 암호를 입력하는 사용자는 인증되지 않습니다. 하지만 이 경우에도 해커가 UserName 텍스트 상자에 겉보기에는 잘못된 것이 없는 다음과 같은 텍스트를 입력하여 유효한 사용자 이름 및 암호를 알지 못하더라고 시스템에 침입할 수 있습니다. 'Or 1=1 --해커는 잘못 형성된 SQL을 쿼리에 주입하여 시스템에 침입합니다. 이 경우의 해킹은 다음과 같이 사용자가 입력한 고정 문자열 및 값의 연결을 통해 실행 쿼리가 형성되기 때문에 작동됩니다. String strQry = "SELECT Count(*) FROM Users WHERE USerName='"+ txtUser.Text + "' AND PassWord='" + txtPassword.Text + "'"; 유효한 사용자 이름인 "Paul"과 암호 "password"를 사용자가 입력하는 경우 strQry는 다음과 같이 됩니다. SELECT Count(*) FROM Users WHERE UserName='Paul' AND Password='password' 하지만 해커가 다음을 입력하면 ' Or 1=1 - 쿼리가 다음과 같이 되니다. SELECT Cont(*) FROM Users WHERE UserName="Or 1=1 -' AND Password="

이중 하이픈은 SQL 에서 주석의 시작 부분을 나타내므로 쿼리는 다음과 같이 됩니다.

SELECT Count (*) FROM Users WHERE UserName=" Or 1=1 식 1=1은 테이블의 모든 행에 대해 항상 True이고 다른 식이 포함된 True 식 or'd는 항상 0 이 아님 레코드 개수를 반환합니다.

일부 SQL injections 공격에는 폼 인증이 포함되지 않습니다. 폼 인증과 관련한 SQL injections 공격에 필요한 사항은 동적으로 구성된 일부 SQL과 트러스트되지 않은 사용자 입력이 있는 응용 프로그램입니다. 정확한 조건만 주어진다면 이러한 공격으로 인한 피해 범위를 해커의 SQL 언어 및 데이터베이스 구성에 대한 지식 수준으로만 제한할 수 있습니다.

이제 badProductList.aspx에서 가져온아래 " 표2 "에 표시된 코드를 살펴보시면, 이 페이지는 Northwind 데이터베이스의 제품을 표시하고 사용자가 txtFilter라는 텍스트 상자를 사용하여 제품 결과목록을 필터링하도록 할 수 있습니다. 마지막 예에서와 같이 이 페이지는 실행 SQL이 사용자가 입력하는 값으로 동적으로 생성되기 때문에 SQL injections 공격에 당할 가능성이 높습니다. 이러한 특정 페이지는 약삭빠른 해커가 공격하여 기밀 정보를 훔치고, 데이터베이스의 데이터를 변경하고, 데이터베이스 레코드를 손상시키고, 심지어는 새로운 데이터베이스 사용자 계정을 만들 수도 있기 때문에 해커에게는 천국과도 같습니다.

SQL Server를 포함한 대부분의 SQL 호환 데이터베이스는 메타데이터를 sysobjects, syscolumns, sysindexes 등의 이름으로 일련의 시스템 테이블에 저장합니다. 즉, 해커는 이러한 시스템 테이블을 사용하여 데이터베이스에 대한 스키마 정보를 확신하고 추가적인 데이터베이스 손상을 위한 도움을 얻을 수 있습니다. 예를 들어 다음과 같이 txtFilter 텍스트 상자에 입력된 텍스트는 데이터베이스에서 사용자 테이블의 이름을 확인하는 데 사용될 수 있습니다. ' UNION SELECT id, name, '', 0 FROM sysobjects WHERE xtype ='U' --

```
private void cmdFilter_Click(object sender, System.EventArgs e) {
      dgrProducts.CurrentPageIndex = 0;
      bindDataGrid();
}

private void bindDataGrid() {
      dgrProducts.DataSource = createDataView();
      dgrProducts.DataBind();
}

private DataView createDataView()   {
      string strCnx =
       "server=localhost;uid=sa;pwd=;database=northwind;";
      string strSQL = "SELECT ProductId, ProductName, " +
       "QuantityPerUnit, UnitPrice FROM Products";

      //This code is susceptible to SQL injection attacks.
      if (txtFilter.Text.Length > 0) {
            strSQL += " WHERE ProductName LIKE '" + txtFilter.Text + "'";
      }

      SqlConnection cnx   = new SqlConnection(strCnx);
      SqlDataAdapter sda = new SqlDataAdapter(strSQL, cnx);
      DataTable dtProducts = new DataTable();
sda.Fill(dtProducts);

      return dtProducts.DefaultView;
}
```

표 2

UNION 문은 해커가 한 쿼리의 결과를 다른 쿼리로 분할할 수 있도록 하기 때문에 해커에게 특히 유용합니다. 이러한 경우 해커는 데이터베이스의 사용자 테이블 이름을 제품 테이블의 원래 쿼리로 분할합니다. 여기에 사용된 방법은 단지 열의 개수와 데이터 형식을 원래의 쿼리와 일치시키는 것 뿐입니다. 이전 쿼리는 Users라는 테이블이 데이터베이스에 있음을 나타낼 수 있습니다. 두 번째 쿼리는 Users 테이블에 있는 열을 노출시킬 수 있습니다. 해커는 이러한 정보를 사용하여 txtFilter 텍스트 상자에 다음을 입력할 수 있습니다. ' UNION SELECT 0, UserName, Password, 0 FROM Users -- 이 쿼리를 입력하면 아래그림 1과 같이 Users 테이블에 있는 사용자 이름 및 암호를 노출시킵니다.

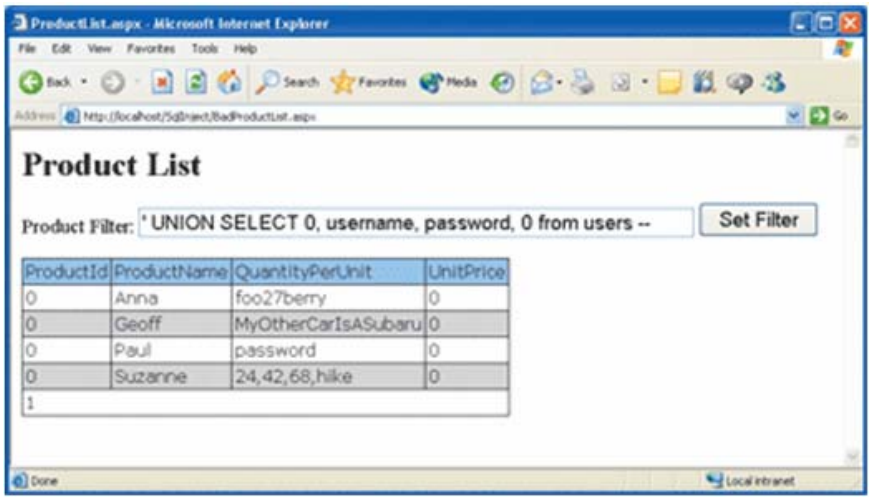

그림 1

SQL injections 공격은 또한 데이터를 변경하거나 데이터베이스를 손상시키는 데에도 사용될 수 있습니다. SQL injections 해커는 txtFilter 텍스트 상자에 다음을 입력하여 첫 번째 제품의 가격을 $18에서 $0.01로 바꾸고 이러한 사실을 다른 사람이 눈치채기 전에 일부 제품을 재빠르게 구매할 수 있습니다. '; UPDATE Products SET UnitPrice = 0.01 WHERE ProductId = 1--

이러한 해킹은 SQL Server에서 세미콜론이나 공백을 사용하여 구분된 여러 SQL 문을 함께 입력할 수 있도록 허용하기 때문에 가능합니다. 이 예에서 DataGrid는 아무 것도 표시하지 않지만 업데이트 쿼리는 성공적으로 실행됩니다. 이러한 같은 기술을 사용하면 DROP TABLE 문을 실행하거나 새로운 사용자 계정을 만들고 이 사용자를 sysadmin 역할에 추가하는 시스템 저장 프로시저를 실행할 수도 있습니다. 이러한 해킹은 모두 “표 2”에 표시된 BadProductList.aspx 페이지를 사용하여 가능합니다.

2. 동일한 해킹 기회

SQL injections 공격은 SQL Server에만 국한된 문제가 아닙니다. Oracle, MySQL, DB2, Sybase 등의 다른데이터베이스에서도 이러한 종류의 공격을 받을 수 있습니다. SQL injections 공격은 SQL 언어에 다음과 같이 강력하고 유연한 여러 기능이 포함되어 있기 때문에 가능합니다.

- 이중 하이픈을 사용하여 SQL 문에 주석을 포함시킬 수 있는 기능
- 여러 SQL 문을 함께 입력하고 이를 일괄 처리로 실행할 수 있는 기능
- SQL을 사용하여 표준 시스템 테이블 집합으로부터 메타데이터를 쿼리할 수 있는 기능

일반적으로 데이터베이스에서 지원되는 SQL 언어의 기능이 강력할수록 데이터베이스에 대한 공격 가능성도 높아집니다. 따라서 SQL Server가 injections 공격의 일반적인 대상이 되는 것입니다.

SQL injections 공격은 ASP.NET 응용 프로그램으로만 제한되지 않습니다. 기존의 ASP, Java, JSP 및 PHP 응용 프로그램도 모두 같은 위험이 있습니다. 실제로 SQL injections 공격은 데스크톱 응용 프로그램에 대해서도 수행될 수 있습니다. 예를 들어 이 문서에 대한 다운로드 파일(이 문서의 맨 위에 있는 링크로 제공)에 SQL injections 공격을 받을 수 있는

SQLInjectWinForm이라는 Windows Forms 응용 프로그램 예제를 포함되어 있습니다.

SQL injections 공격 방지를 위한 한 두 개의 핵심 방법을 쉽게 설명할 수도 있지만 이 문제에 대해서는 계층적 방식을 사용하는 것이 가장 좋습니다. 이러한 방식에서는 일부 취약성으로 인해 보안 방식 중 하나가 무효화되더라도 계속해서 보호 상태를 유지할 수 있습니다. 권장되는 계층은 아래 그림과 같습니다.

Principle	Implementation
Never trust user input	Validate all textbox entries using validation controls, regular expressions, code, and so on
Never use dynamic SQL	Use parameterized SQL or stored procedures
Never connect to a database using an admin-level account	Use a limited access account to connect to the database
Don't store secrets in plain text	Encrypt or hash passwords and other sensitive data; you should also encrypt connection strings
Exceptions should divulge minimal information	Don't reveal too much information in error messages; use customErrors to display minimal information in the event of unhandled error; set debug to false

그림 2

3. 모든 입력에 대한 검사 수행

위 그림에 나열된 첫 번째 원칙은 매우 중요한 것입니다. 모든 사용자 입력은 악의적인 것으로 간주하십시오! 데이터베이스 쿼리에서 검사되지 않은 사용자 입력을 사용해서는 안됩니다. 특히 RegularExpressionValidator 컨트롤과 같은 ASP.NET 유효성 검사 컨트롤은 사용자 입력의 유효성을 검사하기 위한 훌륭한 도구입니다.

유효성 검사에 대한 기본적인 두 가지 방식은 문제가 있는 문자를 허용하지 않거나 적은 수의 필수 문자만 허용하는 것입니다. 하이픈과 작은따옴표와 같은 문제가 되는 일부 문자를 쉽게 허용하지 않을 수도 있지만 이 방법은 두 가지 이유로 인해 적합하지 않을 수 있습니다. 첫 번째, 해커에게 유용하게 사용되는 문자를 놓칠 수 있으며, 두 번째 잘못된 문자를 표현하는 방법이 여러 가지일 수 있습니다. 예를 들어 해커는 작은따옴표를 이스케이프 처리하여 유효성 검사 코드에서 놓치도록 만들거나 이스케이프 처리된 따옴표를 데이터베이스에 전달하여 일반적인 작은따옴표 문자와 동일하게 취급되도록 할 수 있습니다. 더 나은 방법은 허용 가능한 문자를 식별하고 해당 문자만 허용하는 것입니다. 이러한 방식에는 더 많은 작업이 필요하지만 입력에 대해 보다 세밀한 제어가 가능하며 보다 안전합니다. 어떤 방식을 사용하던 간에 일부 해킹에는 많은 수의 문자가 필요하므로 입력에 대한 길이를 제한할 수 있습니다.

GoodLogin.aspx(다운로드 코드에서 제공)에는 두 개의 일반 식 유효성 검사 컨트롤이 포함되며, 이 중에서 하나는 사용자 이름에 대한 컨트롤이고 다른 하나는 암호에 대한 컨트롤입니다. 또한 여기에는 4-12개의 숫자, 알파벳 문자 및 밑줄로 입력을 제한하는 다음과 같은 ValidationExpression 값이 포함되어 있습니다. [₩d_a-zA-Z]{4,12}

사용자가 텍스트 상자에 잠재적으로 손상될 가능성이 있는 문자를 입력하도록 허용해야 할

수 있습니다. 예를 들어 사용자 이름의 일부로 작은따옴표(또는 어포스트로피)를 입력해야 할 수 있습니다. 이러한 경우 정규 식이나 String.Replace 메서드를 사용하여 각각의 작은따옴표를 두 개의 작은따옴표로 바꾸면 작은따옴표를 안전하게 렌더링할 수 있습니다. 예를 들면 다음과 같습니다. string strSanitizedInput = strInput.Replace("'", "''");

4. 동적 SQL 방지

이 문서에서 설명하는 SQL injections 공격은 모두 동적 SQL의 실행을 기반으로 합니다. 즉, 사용자가 입력한 값과 SQL을 연결하여 생성되는 SQL 문입니다. 하지만 매개 변수가 있는 SQL을 사용하면 해커가 SQL을 코드에 주입할 수 있는 가능성이 크게 줄어듭니다.

```
private void cmdLogin_Click(object sender, System.EventArgs e) {
    string strCnx = ConfigurationSettings.AppSettings["cnxNWindBad"];
    using (SqlConnection cnx = new SqlConnection(strCnx))
    {
        SqlParameter prm;
        cnx.Open();
        string strQry =
            "SELECT Count(*) FROM Users WHERE UserName=@username " +
            "AND Password=@password";
        int intRecs;
        SqlCommand cmd = new SqlCommand(strQry, cnx);
        cmd.CommandType= CommandType.Text;
        prm = new SqlParameter("@username",SqlDbType.VarChar,50);
        prm.Direction=ParameterDirection.Input;
        prm.Value = txtUser.Text;
        cmd.Parameters.Add(prm);
        prm = new SqlParameter("@password",SqlDbType.VarChar,50);
        prm.Direction=ParameterDirection.Input;
        prm.Value = txtPassword.Text;
        cmd.Parameters.Add(prm);
        intRecs = (int) cmd.ExecuteScalar();
        if (intRecs>0) {
            FormsAuthentication.RedirectFromLoginPage(txtUser.Text, false);
        }
        else {
            lblMsg.Text = "Login attempt failed.";
        }
    }
}
```

표 3

위의 표3의 코드는 매개 변수가 있는 SQL을 사용하여 injections 공격을 방지합니다. 매개 변수가 있는 SQL은 사용자가 임시 SQL을 사용해야 하는 경우 뛰어난 성능을 보여 줍니다. 이러한 방식은 IT 부서에서 저장 프로시저를 믿지 않거나 버전 5.0까지 이를 지원하지 않은 MySQL과 같은 제품을 사용하는 경우에 필수적입니다. 하지만 가능하다면 추가 기능에 대해 저장 프로시저를 사용하여 데이터베이스에 있는 기본 테이블에 대한 모든 권한을 제거하여 그림 1에 표시된 것과 같은 쿼리를 만들 수 있는 가능성을 제거해야 합니다.

```
private void cmdLogin_Click(object sender, System.EventArgs e) {
    string strCnx =
        ConfigurationSettings.AppSettings["cnxNWindBetter"];
    using (SqlConnection cnx = new SqlConnection(strCnx))
    {
        SqlParameter prm;

        cnx.Open();

        string strAccessLevel;

        SqlCommand cmd = new SqlCommand("procVerifyUser", cnx);
        cmd.CommandType= CommandType.StoredProcedure;

        prm = new SqlParameter("@username",SqlDbType.VarChar,50);
        prm.Direction=ParameterDirection.Input;
        prm.Value = txtUser.Text;
        cmd.Parameters.Add(prm);

        prm = new SqlParameter("@password",SqlDbType.VarChar,50);
        prm.Direction=ParameterDirection.Input;
        prm.Value = txtPassword.Text;
        cmd.Parameters.Add(prm);

        strAccessLevel = (string) cmd.ExecuteScalar();

        if (strAccessLevel.Length>0) {
            FormsAuthentication.RedirectFromLoginPage(txtUser.Text, false);
        }
        else {
            lblMsg.Text = "Login attempt failed.";
        }
    }
}
```

표 4

표 4에 표시된 BetterLogin.aspx는 procVerifyUser라는 저장 프로시저를 사용하여 사용자에 대한 유효성을 검사합니다.

5. 최소 권한으로 실행

BadLogin.aspx 및 BadProductList.aspx에서 보여진 잘못된 구현 방법 중 하나는 sa 계정을 통해 연결 문자열을 사용한다는 점입니다. 다음은 Web.config에서 발견할 수 있는 연결 문자열입니다.

<add key="cnxNWindBad" value="server=localhost;uid=sa;pwd=;database=northwind;" />

이 계정은 로그인 생성과 데이터베이스 삭제 등 거의 모든 작업을 수행할 수 있는 System Administrators 역할로 실행됩니다. 말할 필요도 없이 응용 프로그램 데이터베이스 액세스

에 대해 sa(또는 고급 권한이 있는 계정)를 사용하는 것은 매우 잘못된 생각입니다. 대신 제한된 액세스 계정을 만들고 이를 사용하도록 하는 것이 훨씬 좋습니다. GoodLogin.aspx에 사용된 계정은 다음과 같은 연결 문자열을 사용합니다. <add key="cnxNWindGood" value="server=localhost;uid=NWindReader;pwd=utbbeesozg4d; database=northwind;" />

NWindReader 계정은 db_datareader 역할에 따라 실행되며 이 역할은 데이터베이스에 대한 테이블의 읽기로 액세스를 제한합니다. BetterLogin.aspx는 저장 프로시저와 WebLimitedUser 로그인을 사용하여 상황을 더욱 향상시켜 줍니다. 이 로그인에는 해당 저장 프로시저를 실행하는 권한만 포함되어 있으며 기본 테이블에 대해서는 어떠한 권한도 없습니다.

6. 기밀 정보를 안전하게 저장하기

그림 1에 표시된 SQL injections 공격은 Users 테이블의 사용자 이름과 암호를 노출시킵니다. 이러한 종류의 테이블은 일반적으로 폼 인증을 적용하는 경우에 사용되며 대부분의 응용 프로그램에서 암호는 일반 텍스트로 저장됩니다. 더 나은 방법은 데이터베이스에서 암호화된 암호 또는 해시된 암호를 저장하는 것입니다. 해시된 암호는 암호를 해독할 수 없기 때문에 암호화된 암호보다 안전합니다. 해시에 salt(암호화 방식으로 안전한 임의 값)를 추가하여 해시된 암호의 보안 성능을 더욱 높일 수도 있습니다. BestLogin.aspx에는 사용자가 입력한 암호를 SecureUsers 테이블에 저장된 암호의 salt로 지정된 해시된 암호와 비교하는 코드가 포함되어 있습니다(표 5 참조).

해시된 퍼즐에 대한 또 다른 측면은 AddSecureUser.aspx에서 볼 수 있습니다. 이 페이지를 사용하면 salt로 지정되어 해시된 암호를 생성하고 이를 SecureUsers 테이블에 저장할 수 있습니다. BestLogin.aspx 및 AddSecureUser.aspx는 모두 “표 6”에 표시된 것과 같이 SaltedHash 클래스 라이브러리의 코드를 사용합니다. Jeff Prosise가 만든 이 코드는 System.Web.Security 네임스페이스의 FormsAuthentication.HashPasswordForStoringInConfigFile 메서드를 사용하여 암호 해시를 만들고 System.Security.Cryptography 네임스페이스의 RNGCryptoServiceProvider.GetNonZeroBytes 메서드를 사용하여 16바이트의 임의 salt 값(Convert.ToBase64String을 사용하여 문자열로 변환할 겨우 24 문자가 됨)을 만들 수 있습니다.

SQL injections 공격과 직접 관련되지는 않지만 BestLogin.aspx는 연결 문자열의 암호화라는 또 다른 최선의 보안 구현 방법을 보여 줍니다. 연결 문자열은 포함된 데이터베이스 계정 암호가 들어 있는 경우 BestLogin.aspx에서와 같이 특히 중요하게 보호되어야 합니다. 데이터베이스에 연결하려면 연결 문자열의 암호를 해독해야 하기 때문에 연결 문자열은 해시할 수 없습니다. 그 대신 연결 문자열을 암호화해야 합니다. 다음은 Web.config에 저장되어 있고 BestLogin.aspx에서 사용되는 암호화된 연결 문자열을 보여 줍니다. <add key="cnxNWindBest" value="AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAAcWMZ8XhPz0O8jHcS1539LAQAAAACAAAAAAADZgAAqAAAABAAAABdodw0YhWfcC6+
UjUUOiMwAAAAAASAAACgAAAAEAAAALPzjTRnAPt7/W8v38ikHL5lAAAAzctRyEcHxWkzxeqbq/ V9ogaSqS4UxvKC9zmrXUoJ9mwrNZ/

XZ9LgbfcDXIIAXm2DLRCGRHMtrZrp9yledz0n9kgP3b3s+

X8wFAAAANmLu0UfOJdTc4WjlQQgmZElY7Z8" />

```
private void cmdLogin_Click(object sender, System.EventArgs e) {
    try {
        // Grab the encrypted connection string and decrypt it
        string strCnx = SecureConnection.GetCnxString("cnxNWindBest");

        // Establish connection to database
        using (SqlConnection cnx = new SqlConnection(strCnx))
        {
            SqlParameter prm;

            cnx.Open();

            // Execute sproc to retrieved hashed password for this user
            string strHashedDbPwd;

            SqlCommand cmd = new SqlCommand("procGetHashedPassword", cnx);
            cmd.CommandType = CommandType.StoredProcedure;
            prm = new SqlParameter("@username", SqlDbType.VarChar,50);
            prm.Direction = ParameterDirection.Input;
            prm.Value = txtUser.Text;
            cmd.Parameters.Add(prm);

            strHashedDbPwd = (string) cmd.ExecuteScalar();

            if (strHashedDbPwd.Length>0) {
                // Verify that hashed user-entered password is the same
                // as the hashed password from the database
                if (SaltedHash.ValidatePassword(txtPassword.Text,
                    strHashedDbPwd)) {
                    FormsAuthentication.RedirectFromLoginPage(
                        txtUser.Text, false);
                }
                else {
                    lblMsg.Text = "Login attempt failed.";
                }
            }
            else {
                lblMsg.Text = "Login attempt failed.";
            }
        }
    }
    catch {
        lblMsg.Text = "Login attempt failed.";
    }
}
```

표 5

```
public class SaltedHash {
    static public bool ValidatePassword (string password,
        string saltedHash) {

        // Extract hash and salt string
        const int LEN = 24;
        string saltString = saltedHash.Substring(saltedHash.Length - LEN);
        string hash1 = saltedHash.Substring(0, saltedHash.Length - LEN);

        // Append the salt string to the password
        string saltedPassword = password + saltString;

        // Hash the salted password
        string hash2 =
            FormsAuthentication.HashPasswordForStoringInConfigFile(
            saltedPassword, "SHA1");

        // Compare the hashes
        return (hash1.CompareTo(hash2) == 0);
    }

    static public string CreateSaltedPasswordHash (string password) {

        // Generate random salt string
        RNGCryptoServiceProvider csp = new RNGCryptoServiceProvider();
        byte[] saltBytes = new byte[16];
        csp.GetNonZeroBytes(saltBytes);
        string saltString = Convert.ToBase64String(saltBytes);

        // Append the salt string to the password
        string saltedPassword = password + saltString;

        // Hash the salted password
        string hash =
            FormsAuthentication.HashPasswordForStoringInConfigFile(
            saltedPassword, "SHA1");

        // Append the salt to the hash
        return hash + saltString;
    }
}
```

표 6

```
public class SecureConnection {
    static public string GetCnxString(string configKey) {
        string strCnx;

        try {
            // Grab encrypted connection string from web.config
            string strEncryptedCnx =
                ConfigurationSettings.AppSettings[configKey];

            // Decrypt the connection string
            DataProtector dp = new
                DataProtector(DataProtector.Store.USE_MACHINE_STORE);
            byte[ ] dataToDecrypt =
                Convert.FromBase64String(strEncryptedCnx);
            strCnx =
                Encoding.ASCII.GetString(dp.Decrypt(dataToDecrypt,null));
        }
        catch {
            strCnx="";
        }

        return strCnx;
    }
}
```

표 7

BestLogin은 “표 7”에서와 같이 SecureConnection 클래스로부터 GetCnxString 메서드를 호출하여 cnxNWindBest AppSetting 값을 검색하고 이를 다음 코드로 암호 해독합니다.
string strCnx = SecureConnection.GetCnxString("cnxNWindBest");

순서대로 SecureConnection 클래스는 호출을 Win32 DPAPI(데이터 보호 API)로 래핑하는 DataProtect 클래스 라이브러리(여기에 표시되지는 않지만 이 문서의 다운로드에 포함되어 있음)를 호출합니다. DPAPI의 다양한 기능 중 하나는 사용자를 위해 암호화 키를 관리하는 것입니다. DataProtect 클래스 라이브러리 및 이 라이브러리를 사용할 때 고려해야 하는 추가 옵션에 대한 자세한 내용은 Microsoft 패턴 및 연습 가이드 중 하나인 "보안적인 ASP.NET 응용 프로그램 작성: 인증, 권한 부여 및 보안 통신 (영문)"을 참조하십시오.
http://msdn.microsoft.com/ko-kr/library/ms994921(en-us).aspx

EncryptCnxString.aspx 페이지를 사용하면 시스템 특정 암호화된 연결 문자열을 만들어서 구성 파일에 붙여 넣을 수 있습니다. 이 페이지는 그림 3에 표시되어 있습니다. 물론, 암호화하거나 해시해야 하는 기밀 정보에는 암호 및 연결 문자열 외에도 신용 카드 번호 및 해커에게 노출될 경우 위험할 수 있는 기타 모든 정보가 포함됩니다. ASP.NET 2.0에는 암호 해싱 및 연결 문자열 암호화를 단순하게 만들어 주는 여러 기능이 포함되어 있습니다.

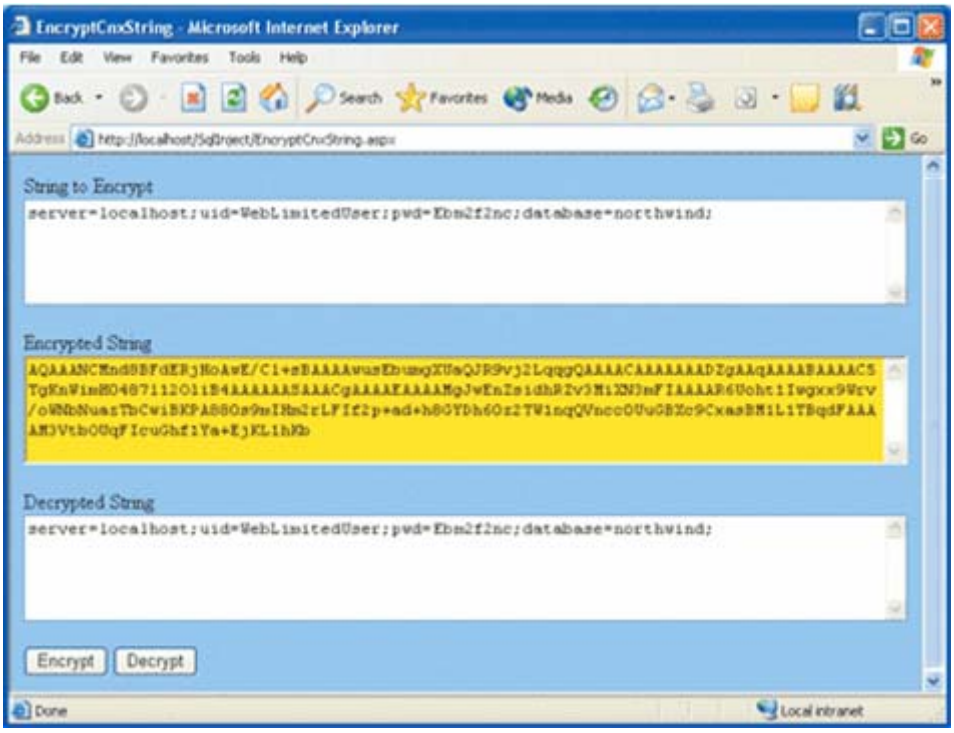

그림 3

7. 안전한 실패

런타임 예외를 적절하게 처리하지 못할 경우에도 해커는 이를 악용할 수 있습니다. 따라서 모든 프로덕션 코드에 예외 처리기를 포함시키는 것이 중요합니다. 또한 처리된 예외 및 처리되지 안은 예외는 항상 해커에게 도움이 될 수 있는 정보를 최소한으로만 제공해야 합니다. 처리된 예외의 경우 오류 메시지에서 원래 사용자에 대한 유용성과 악의적인 해커에게 너무 많은 정보를 제공하지 않는 보안성 간의 균형을 맞추어야 합니다.

처리되지 않은 예외의 경우에는 컴파일 요소(Web.config 파일)의 디버그 특성을 False로 설정하고 customErrors 요소의 모드 특성을 On 또는 RemoteOnly로 설정하여 해커에게 최소한의 정보만 표시되도록 해야 합니다. 예를 들어 다음을 보십시오. <compilation defaultLanguage="c#" debug="false" /> <customErrors mode="RemoteOnly" />

RemoteOnly 설정은 localhost로부터 사이트에 액세스하는 사용자에게는 유용한 오류 메시지를 제공하고 원격 위치로부터 사이트에 액세스하는 다른 사용자에게는 예외에 대한 어떤 유용한 정보도 노출시키지 않는 일반적인 오류 메시지를 제공하도록 보장합니다. 로컬 사용자를 포함한 모든 사용자에게 일반 오류 메시지를 표시하도록 하려면 On 설정을 사용하십시오. 프로덕션 환경에서는 Off 설정을 절대로 사용하지 마십시오.

8. 결론

SQL injections 공격은 보안 시스템에 침입하여 데이터를 훔치거나, 변경 또는 삭제할 우려가 있기 때문에 응용 프로그램 개발자가 신중하게 다루어야 하는 문제입니다. 사용하는 ASP.NET 버전과는 관계없이 이러한 공격에는 너무나 쉽게 취약해질 수 있습니다. 실제로 ASP.NET을 사용하지 않는 경우에도 SQL injections 공격에 쉽게 당할 수 있습니다. Windows Forms 응용 프로그램과 같이 사용자 입력 데이터를 사용하여 데이터베이스를 쿼

리하는 모든 응용 프로그램은 잠재적으로 injections 공격의 대상이 될 수 있습니다.

SQL injections 공격으로부터 자신을 보호하는 방법은 그렇게 어렵지 않습니다. 모든 사용자 입력에 대한 유효성을 검사 및 확인하고, 동적 SQL을 절대로 사용하지 않고, 최소 권한으로 계정을 사용하고, 해당 기밀 정보를 해시 또는 암호화하고, 해커에게 유용한 정보를 제공하지 않도록 거의 정보를 표시하지 않는 오류 메시지를 제공하는 응용 프로그램은 SQL injections 공격을 매우 효과적으로 방지할 수 있습니다. 여러 계층의 공격 방어 방법을 사용할 경우 하나의 방법이 실패하더라도 계속해서 보호 상태를 유지할 수 있습니다.