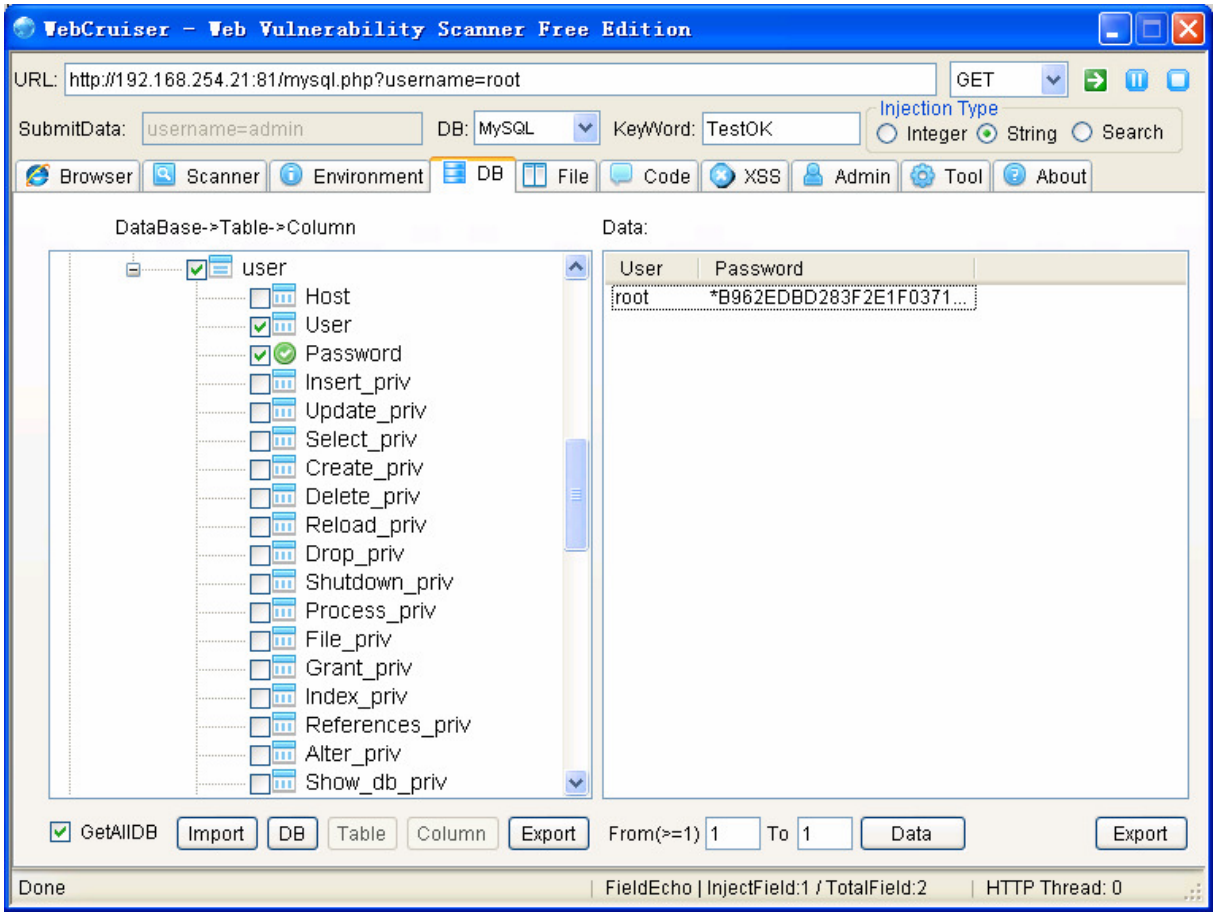
SQL Injection

1. What is SQL Injection?

SQL injection is a code injection technique that exploits a security vulnerability occurring in the database layer of an application. The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed. It is an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another. SQL Injection is one of the most common application layer attack techniques used today.
Here is a POC(Proof of Concept) by Scanning Tool: WebCruiser - Web Vulnerability Scanner:



2. Forms of vulnerability

2.1. Incorrectly filtered escape characters

This form of SQL injection occurs when user input is not filtered for escape characters and is then passed into an SQL statement. This results in the potential manipulation of the statements performed on the database by the end user of the application.
The following line of code illustrates this vulnerability:

```
statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```

This SQL code is designed to pull up the records of the specified username from its table of users. However, if the "userName" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended. For example, setting the "userName" variable as

```
a' or 't'='t
```

renders this SQL statement by the parent language:

```
SELECT * FROM users WHERE name = 'a' OR 't'='t';
```

If this code were to be used in an authentication procedure then this example could be used to force the selection of a valid username because the evaluation of 't'='t' is always true.
The following value of "userName" in the statement below would cause the deletion of the "users" table as well as the selection of all data from the "userinfo" table (in essence revealing the information of every user), using an API that allows multiple statements:

```
a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't
```

This input renders the final SQL statement as follows:

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT *
FROM userinfo WHERE 't' = 't';
```

While most SQL server implementations allow multiple statements to be executed with one call in this way, some SQL APIs such as PHP's mysql_query() do not allow this for security reasons. This prevents attackers from injecting entirely separate queries, but doesn't stop them from modifying queries.

2.2. Incorrect type handling

This form of SQL injection occurs when a user supplied field is not strongly typed or is not checked for type constraints. This could take place when a numeric field is to be used in a SQL statement, but the programmer makes no checks to validate that the user supplied input is numeric. For example:

```
statement := "SELECT * FROM userinfo WHERE id = " + a_variable + ";"
```

It is clear from this statement that the author intended a_variable to be a number correlating to the "id" field. However, if it is in fact a string then the end user may manipulate the statement as they choose, thereby bypassing the need for escape characters. For example, setting a_variable to

```
1;DROP TABLE users
```

will drop (delete) the "users" table from the database, since the SQL would be rendered as follows:

```
SELECT * FROM userinfo WHERE id=1;DROP TABLE users;
```

2.3. Vulnerabilities inside the database server

Sometimes vulnerabilities can exist within the database server software itself, as was the case with the MySQL server's mysql_real_escape_string() function. This would allow an attacker to perform a successful SQL injection attack based on bad Unicode characters even if the user's input is being escaped. This bug was patched with the release of version 5.0.22 (released on 24th May 06).

2.4. Blind SQL injection

Blind SQL Injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page. This type of attack can become time-intensive because a new statement must be crafted for each bit recovered. There are several tools that can automate these attacks once the location of the vulnerability and the target information has been established.

2.4.1. Conditional responses

One type of blind SQL injection forces the database to evaluate a logical statement on an ordinary application screen.

```
SELECT booktitle FROM booklist WHERE bookId = 'OOk14cd' AND 1=1;
```

will result in a normal page while

```
SELECT booktitle FROM booklist WHERE bookId = 'OOk14cd' AND 1=2;
```

will likely give a different result if the page is vulnerable to a SQL injection. An injection like this may suggest to the attacker that a blind SQL injection is possible, leaving the attacker to devise statements that evaluate to true or false depending on the contents of another column or table outside of the SELECT statement's column list.

2.4.2. Conditional errors

This type of blind SQL injection causes an SQL error by forcing the database to evaluate a statement that causes an error if the WHERE statement is true. For example,

```
SELECT 1/0 FROM users WHERE username='Ralph';
```

the division by zero will only be evaluated and result in an error if user Ralph exists.

2.4.3. Time delays

Time Delays are a type of blind SQL injection that cause the SQL engine to execute a long running query or a time delay statement depending on the logic injected. The attacker can then measure the time the page takes to load to determine if the injected statement is true.

3. Preventing SQL injection

To protect against SQL injection, user input must not directly be embedded in SQL statements. Instead, parameterized statements must be used (preferred), or user input must be carefully escaped or filtered.

3.1. Parameterized statements

With most development platforms, parameterized statements can be used that work with parameters (sometimes called placeholders or bind variables) instead of embedding user input in the statement. In many cases, the SQL statement is fixed. The user input is then assigned (bound) to a parameter. This is an example using Java and the JDBC API:

```
PreparedStatement prep = conn.prepareStatement("SELECT * FROM
USERS WHERE USERNAME=? AND PASSWORD=?");
prep.setString(1, username);
prep.setString(2, password);
prep.executeQuery();
```

Similarly, in C#:

```
using (SqlCommand myCommand = new SqlCommand("SELECT * FROM
USERS WHERE USERNAME=@username AND
PASSWORD=HASHBYTES('SHA1',
  @password)", myConnection))
    {
        myCommand.Parameters.AddWithValue("@username", user);
```

```
        myCommand.Parameters.AddWithValue("@password", pass);

        myConnection.Open();
        SqlDataReader myReader = myCommand.ExecuteReader())
        .....
    }
```

In PHP version 5 and above, there are multiple choices for using parameterized statements. The PDO[5] database layer is one of them:

```
$db = new PDO('pgsql:dbname=database');
$stmt = $db->prepare("SELECT priv FROM testUsers WHERE username=:username AND password=:password");
$stmt->bindParam(':username', $user);
$stmt->bindParam(':password', $pass);
$stmt->execute();
```

There are also vendor-specific methods; for instance, using the mysqli[6] extension for MySQL 4.1 and above to create parameterized statements:

```
$db = new mysqli("localhost", "user", "pass", "database");
$stmt = $db -> prepare("SELECT priv FROM testUsers WHERE username=? AND password=?");
$stmt -> bind_param("ss", $user, $pass);
$stmt -> execute();
```

In ColdFusion, the CFQUERYPARAM statement is useful in conjunction with the CFQUERY statement to nullify the effect of SQL code passed within the CFQUERYPARAM value as part of the SQL clause.[8][9]. An example is below.

```
<cfquery name="Recordset1" datasource="cafetownsend">
SELECT *
FROM COMMENTS
WHERE COMMENT_ID =<cfqueryparam value="#URL.COMMENT_ID#" cfsqltype="cf_sql_numeric">
</cfquery>
```

3.1.1. Enforcement at the database level

Currently only the H2 Database Engine supports the ability to enforce query parameterization.[10] However, one drawback is that query-by-example may not be possible or practical because it's difficult to implement query-by-example using parametrized queries.

3.1.2. Enforcement at the coding level

Using object-relational mapping libraries avoids the need to write SQL code. The ORM library in effect will generate parameterized SQL statements from

object-oriented code.

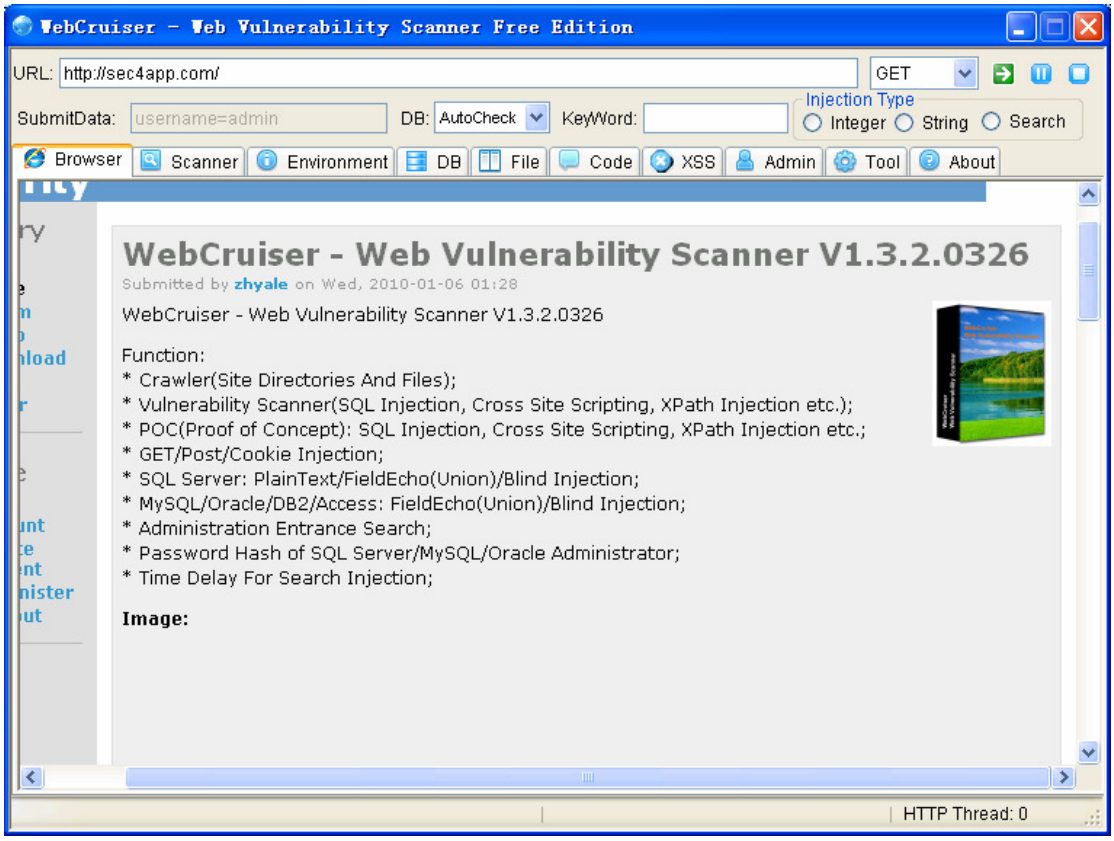
3.2. Escaping

A straight-forward, though error-prone, way to prevent injections is to escape dangerous characters. One of the reasons for it being error prone is that it is a type of blacklist which is less robust than a whitelist. For instance, every occurrence of a single quote (') in a parameter must be replaced by two single quotes ('') to form a valid SQL string literal. In PHP, for example, it is usual to escape parameters using the function mysql_real_escape_string before sending the SQL query:

```
$query = sprintf("SELECT * FROM Users where UserName='%s' and Password='%s'",
                        mysql_real_escape_string($Username),
                        mysql_real_escape_string($Password));
mysql_query($query);
```

3.3. Use Web Vulnerability Scanner

WebCruiser - Web Vulnerability Scanner
WebCruiser - Web Vulnerability Scanner, a compact but powerful web security scanning tool! It has a Crawler and Vulnerability Scanner(SQL Injection, Cross Site Scripting, XPath Injection etc.). It can support not only scanning website, but also Prooving of concept for web vulnerabilities: SQL Injection, Cross Site Scripting, XPath Injection etc. So, WebCruiser is also a SQL Injector, a XPath Injector , and a Cross Site Scripting tool!
You can download it from http://sec4app.com/ .


WebCruiser - Web Vulnerability Scanner Free Edition
URL: http://sec4app.com/
GET
SubmitData: username=admin
DB: AutoCheck
KeyWord:
Injection Type
Integer String Search
Browser Scanner Environment DB File Code XSS Admin Tool About
WebCruiser - Web Vulnerability Scanner V1.3.2.0326
Submitted by zhyale on Wed, 2010-01-06 01:28
WebCruiser - Web Vulnerability Scanner V1.3.2.0326
Function:
* Crawler(Site Directories And Files);
* Vulnerability Scanner(SQL Injection, Cross Site Scripting, XPath Injection etc.);
* POC(Proof of Concept): SQL Injection, Cross Site Scripting, XPath Injection etc.;
* GET/Post/Cookie Injection;
* SQL Server: PlainText/FieldEcho(Union)/Blind Injection;
* MySQL/Oracle/DB2/Access: FieldEcho(Union)/Blind Injection;
* Administration Entrance Search;
* Password Hash of SQL Server/MySQL/Oracle Administrator;
* Time Delay For Search Injection;
Image:
HTTP Thread: 0