

@@@@@@	@@@@@@	@@@	@@@	@@@	@@@	@@@	@@@@@@@@	@@@@@@@@	@@@@@@@@	@@@@	@@@@@@	@@@	@@@
@@@@@@@@	@@@@@@@@	@@@	@@@	@@@@	@@@	@@@	@@@@@@@@	@@@@@@@@	@@@@@@@@	@@@@	@@@@@@@@	@@@@	@@@@
!@@	@@! @@@	@@!	@@!	@@!@@@		@@!	@@!	!@@		@@!	@@!	@@!	@@!@@@
!@!	!@! @!@	!@!	!@!	!@! !@!@!		!@!	!@!	!@!		!@!	!@!	!@!	!@! !@!@!
!!@@!!	@!@ !@!	@!!	!!@	@!@ !!@!		!!@	@!!!!	!@!		@!!	!!@	@!@ !@!	@!@ !!@!
!!@!!!	!@! !!!	!!!	!!!	!@! !!!		!!!	!!!!!!:	!!!		!!!	!!!	!@! !!!	!@! !!!
!;!	!!:!!:!!:	!!:	!!:	!!:		!!:	!!:	:!!		!!:	!!:	!!:	!!:
!;!	:! :!:	:!:	:!:	:!:	!!:	:!:	:!:	:!:		:!:	:!:	:!:	:!:
:::: ::	:::: :!	:: :::	::	::	::	::	::	::	::	::	::	:::: ::	:: ::
:: : :	: : :::	: : : :	:	::	:	:	::	::	:	:	:	: : :	::

SQL (Structured Query Language)

- Database language used for the storage and retrieval of information
- Relational databases
- Information can be interacted with using this declarative language
- Powerful
- Can be manipulated

SQLi - What is it?

- Just like with XSS it is a classic code vs. data problem
- User input is interpreted as code and executed as part of the SQL statement

Example

- A form takes a username and pulls up information on that user
- For takes a name, and if you have permissions to lookup that person, shows you their information
- Code looks like:

```
“SELECT * FROM users WHERE name='%s’” % user_input
```

Example

- If I input “alice” then the query looks like:

```
SELECT * FROM users WHERE name='alice'
```

- How can I pull everyone's records?

Example

- If my name becomes: alice' or '1'='1
- Then the query becomes:

```
SELECT * FROM users WHERE name='alice' or '1'='1'
```

- And all records are returned because 1 will always equal 1

Why it's bad

- Bypass login
- Exfiltrate data
- Elevate privilege
- Tamper with logs/records
- Own the host computer
- Delete everything
- **This is automatically a critical vulnerability**

How to find it

- Supply unexpected user input such as ' ") -- #
- Identify any error messages or changes in response/behavior
- Determine if your input is being executed as code
- Types of searching:
 - Regular – see if extra data is returned
 - Equivalency – see if statements are executed differently
 - Blind – see if you can cause a backend delay or out-of-band response

Testing steps (text data)

- Does the DB send an error back when it receives a ' or " or) or –
- If you get an error, read it
- Does sending " (two single ticks) alleviate the error?
- Test to see if the DB does the same thing when you input FOO as it does when you input:
 - '| |'FOO (Oracle)
 - '+ 'FOO (MS-SQL)
 - ' 'FOO (space between the single ticks) (MySQL)

Testing steps (numerical data)

- Supply a simple mathematical expression
 - If testing for two supply $1+1$ or $3-1$
- User a more complicated expression such as:
 - $67-\text{ASCII}('A')$ $67 - 65 = 2$
 - $51-\text{ASCII}(1)$ $51 - 49 = 2$

Remember

- Certain SQL characters also have special meaning for HTTP so be careful with:
 - & %26
 - = %3d
 - (space) %20
 - + %2b
 - ; %3b

Figure out the DB

- Issue DB specific commands
- Text data:
 - Oracle: 'foo' || 'bar'
 - MS-SQL: 'foo'+'bar'
 - MySQL: 'foo' 'bar'
- Numeric data:
 - Oracle: BITAND(1,1)-BITAND(1,1)
 - MS-SQL: @@PACK_RECIEVED-@@PACK_RECIEVED
 - MySQL: CONNECTION_ID()-CONNECTION_ID()

Blind

- Cause a noticeable delay:
 - MS-SQL: a' WAITFOR DELAY '00:00:05
 - MySQL: a' sleep(5000)

Note

- Sometimes you need to comment out the rest of the statement:
 - Oracle: -- or /*
 - MS-SQL: --
 - MySQL: # or /*
 - SQLite: -- or /*
 - PostgreSQL: --

Mitigation

- Parameterized Queries (aka prepared statements)
- First define the SQL code, then pass in the parameters later
- Allows the database to distinguish between code and data, regardless of what user input is supplied
- Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker

Mitigation Example – ASP.NET

```
string sql = "SELECT * FROM Customers WHERE CustomerId = @CustomerId";  
SqlCommand command = new SqlCommand(sql);  
command.Parameters.Add(new SqlParameter("@CustomerId",  
System.Data.SqlDbType.Int));  
command.Parameters["@CustomerId"].Value = 1;
```


Mitigation Example - Ruby

```
insert_new_user = db.prepare "INSERT INTO users (name,  
age, gender) VALUES (?, ? ,?)"  
insert_new_user.execute 'aizatto', '20', 'male'
```

Mitigation Example - Java

```
String custname = request.getParameter("customerName");  
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";  
PreparedStatement pstmt = connection.prepareStatement(query);  
pstmt.setString(1, custname);  
ResultSet results = pstmt.executeQuery();
```

Resources

- https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet
- <http://blog.codinghorror.com/give-me-parameterized-sql-or-give-me-death/>
- <http://pentestmonkey.net/category/cheat-sheet/sql-injection>
- <http://www.unixwiz.net/techtips/sql-injection.html>