

Injection Flaws



SQL Injection

- Lack of query parameterization can be exploited and used to execute arbitrary queries against back-end databases
- New malicious commands are added to application, hence the term “injection”
- Occurs when malicious untrusted input is used within SQL queries being executed against back-end application databases
- Injected SQL queries will run under the context of the application account, allowing read and/or write access to application data and even schema!

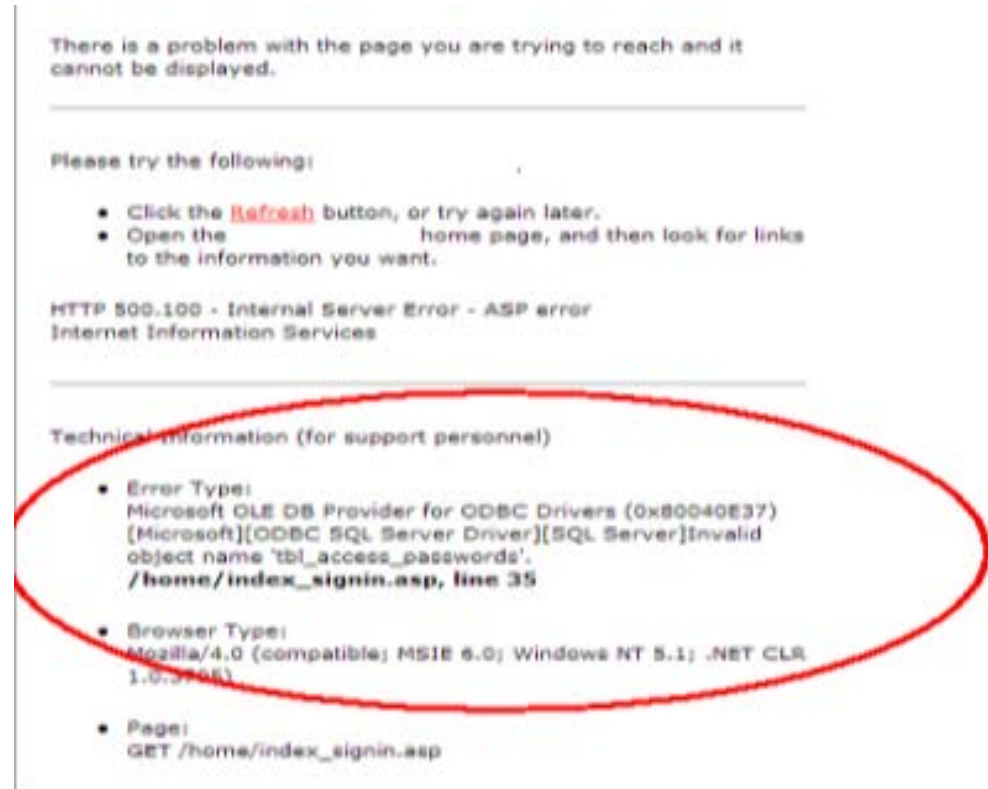
SQL Injection Attack Types

- **Data Retrieval**: Allow an attacker to extract data from the database. Exploits can include modifying the record selection criteria of the SQL query or appending a user-specified query using the SQL UNION directive. This type of exploit can also be used to bypass poorly designed login mechanisms.
- **Data Modification**: Allow attacker to write to database tables. Can be used to modify or add records to the database. (*NOTE: Very dangerous and could result in data corruption!*)
- *DML*
- **Database-Specific Exploits**: Involve exploiting database-specific functionality. Can potentially be used to execute arbitrary commands on the database server operating system. (Command Injection)

SQL Error Messages

■ Where to find error messages?

- To see raw error messages you must **uncheck** Internet Explorer's default setting (Tools -> Internet Options->Advanced):
Show friendly HTTP error messages.



There is a problem with the page you are trying to reach and it cannot be displayed.

Please try the following:

- Click the **Refresh** button, or try again later.
- Open the **home** page, and then look for links to the information you want.

HTTP 500.100 - Internal Server Error - ASP error
Internet Information Services

Technical information (for support personnel)

- **Error Type:**
Microsoft OLE DB Provider for ODBC Drivers (0x80040E37) [Microsoft][ODBC SQL Server Driver][SQL Server]Invalid object name 'tbl_access_passwords'.
/home/index_signin.asp, line 35
- **Browser Type:**
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.0.3705)
- **Page:**
GET /home/index_signin.asp

Anatomy of SQL Injection Attack

```
sql = "SELECT * FROM user_table WHERE username  
= \" & Request("username") & \"' AND password =  
\" & Request("password") & \"'"
```

What the developer intended:

username = chip


password = password

SQL Query:

```
SELECT * FROM user_table WHERE username =  
'john' AND password = 'password'
```

Anatomy of SQL Injection Attack

```
sql = "SELECT * FROM user_table WHERE username  
= ' " & Request("username") & "' AND password =  
' " & Request("password") & "' "
```

 (This is DYNAMIC sql - Bad)

What the developer did not intend is parameter values like:

```
username = john  
password = blah' or '1'='1
```

SQL Query:

```
SELECT * FROM user_table WHERE username =  
'john' AND password = 'blah' or '1'='1'
```

Since **1=1** is true and the AND is executed before the OR, all rows in the users table are returned!

SQL Injection without a Single Quote (')

- Attacks can occur even when variables are not encapsulated within single quotes

```
sql = "SELECT * from users where  
custnum=" +  
request.getParameter("AccountNum");
```

- What happens if AccountNum is **1=1** or **<boolean True>** above?
- Called "Numeric SQL Injection"

String Building to Call Stored Procedures

- String building can be done when calling stored procedures as well

```
sql = "GetCustInfo @LastName=" +  
request.getParameter("LastName");
```

- Stored Procedure Code

```
CREATE PROCEDURE GetCustInfo (@LastName VARCHAR(100))  
AS  
    exec('SELECT * FROM CUSTOMER WHERE LNAME=''' + @LastName + ''')
```

```
GO
```



(Wrapped Dynamic SQL)

- What's the issue here.....

- ▶ If **blah' OR '1'='1** is passed in as the LastName value, the entire table will be returned
- Remember Stored procedures need to be implemented safely. 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation.

Identifying SQL Injection Points

- Insert a single apostrophe into application inputs to invoke a database syntax error
- If a single apostrophe causes a generic error to be returned, SQL injection may still be possible. Modify the string to eliminate the syntax error to validate that a database error is occurring
 - ▶ `blah'--`
 - ▶ `blah' OR '1'='1`
 - ▶ `blah' OR '1'='2`
 - ▶ `Blah'%20'OR%20'1'='1`
 - ▶ `Blah' OR 11;#`
- Trace all application input through the code to see which inputs are ultimately used in database calls
- Identify database calls using SQL string building to check for proper input validation

Code Review: Source and Sink

```
public void bad(HttpServletRequest request, HttpServletResponse response) throws Throwable
{
    String data;

    Logger log_bad = Logger.getLogger("local-logger");

    /* read parameter from request */
    data = request.getParameter("name"); ← Input from request (Source)

    Logger log2 = Logger.getLogger("local-logger");

    Connection conn_tmp2 = null;
    Statement sqlstatement = null;
    ResultSet sqlrs = null;

    try {
        conn_tmp2 = IO.getDBConnection();
        sqlstatement = conn_tmp2.createStatement();

        /* POTENTIAL FLAW: take user input and place into dynamic sql query */
        sqlrs = sqlstatement.executeQuery("select * from users where name='"+data+"'");

        IO.writeString(sqlrs.toString());
    }
    catch( SQLException se )
    {
```

Exploit is executed (Sink)

Code Review: Find the Vulns!

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
{
    String name = req.getParameter("username");
    String pwd = req.getParameter("password");
    int id = validateUser(username, password);
    String retstr = "User : " + name + " has ID: " + id;
    res.getOutputStream().write(retstr.getBytes());
}
```

```
private int validateUser(String user, String pwd) throws Exception
{
    Statement stmt = myConnection.createStatement();
    ResultSet rs;
    rs = stmt.executeQuery("select id from users where
user='" + user + "' and key='" + pwd + "'");
    return rs.next() ? rs.getInt(1) : -1;
}
```

Defending Against SQL Injection

- Validation using Known Good Validation should be used for all input used in SQL queries
- .NET's parameterized queries are extremely resilient to SQL injection attacks, even in the absence of input validation
- Similar functionality exists for Java via PreparedStatements and CallableStatements
 - ▶ Automatically limits scope of user input - cannot break out of variable scope (i.e. it does the escaping for you)
 - ▶ Performs data type checking on parameter values
- Every web language has an API for

Best Practice: Parameterized Queries

Parameterized Queries ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker.

Java EE - use PreparedStatement()

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
```

```
PreparedStatement pstmt = connection.prepareStatement( query );  
pstmt.setString( 1, custname); ResultSet results = pstmt.executeQuery( );
```

Safe C# .NET Prepared Statement Example

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ?";  
try  
{  
OleDbCommand command = new OleDbCommand(query, connection);  
command.Parameters.Add(new OleDbParameter("customerName", CustomerName Name.Text));  
OleDbDataReader reader = command.ExecuteReader(); // ...  
}  
catch (OleDbException se) {  
// error handling  
}
```

.NET Parameterized Query

Dynamic SQL: (Not so Good)

```
string sql = "SELECT * FROM User WHERE Name = '" + NameTextBox.Text  
+ "' AND Password = '" + PasswordTextBox.Text + "'";
```

Parameterized Query: (Nice, Nice!)

```
SqlConnection objConnection = new SqlConnection(_ConnectionString);  
objConnection.Open();  
SqlCommand objCommand = new SqlCommand(  
    "SELECT * FROM User WHERE Name = @Name AND Password =  
    @Password", objConnection);  
objCommand.Parameters.Add("@Name", NameTextBox.Text);  
objCommand.Parameters.Add("@Password", PasswordTextBox.Text);  
SqlDataReader objReader = objCommand.ExecuteReader();  
if (objReader.Read()) { ...
```

Java Prepared Statement



Dynamic SQL: (Not so Good)

```
String sqlQuery = "UPDATE EMPLOYEES SET SALARY = ` +  
    request.getParameter("newSalary") + ` WHERE ID = ` +  
    request.getParameter("id") + `";
```

PreparedStatement: (Nice)

```
double newSalary = request.getParameter("newSalary") ;  
int id = request.getParameter("id");  
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES  
    SET SALARY = ? WHERE ID = ?");  
pstmt.setDouble(1, newSalary);  
pstmt.setInt(2, id);
```

Best Practice: Parameterized Queries

Unsafe HQL Statement Query (Hibernate)

```
unsafeHQLQuery = session.createQuery("from Inventory where productID='"+userSuppliedParameter+"'");
```

Safe version of the same query using named parameters

```
Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid");  
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

Language specific recommendations:

- Java EE - use PreparedStatement() with bind variables
- .NET - use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
- PHP - use PDO with strongly typed parameterized queries (using bindParam())
- Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)

Best Practice: Parameterized Queries

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;
```

RUBY – Active Record

```
# Create
Project.create!(:name => 'owasp')
# Read
Project.all(:conditions => "name = ?", name)
Project.all(:conditions => { :name => name })
Project.where("name = :name", :name => name)
# Update
project.update_attributes!(:name => 'owasp')
# Delete
Project.delete(:name => 'name')
```

Best Practice: Parameterized Queries

Cold Fusion

```
<cfquery name = "getFirst" dataSource = "cfsnippets">  
    SELECT * FROM #strDatabasePrefix#_courses WHERE intCourseID =  
    <cfqueryparam value = #intCourseID# CFSQLType = "CF_SQL_INTEGER">  
</cfquery>
```

Perl - DBI

```
my $sql = "INSERT INTO foo (bar, baz) VALUES ( ?, ? )";  
  
my $sth = $dbh->prepare( $sql );  
  
$sth->execute( $bar, $baz );
```

SQL Injection - Lab/Demo

The screenshot shows a web browser window titled "Damn Vulnerable Web App (DVWA) v1.0.7 :: Vulnerability: SQL Injection - Windows Internet Explorer". The address bar shows the URL "http://127.0.0.1/dvwa/vulnerabilities/sqli/". The browser's menu bar includes "File", "Edit", "View", "Favorites", "Tools", and "Help". The Favorites bar contains "OWASP Application Security...", "Suggested Sites", "Web Slice Gallery", "dvwa - Damn Vulnerable We...", and "SQL Injection Cheat Sheet".

The main content area features the DVWA logo at the top. Below it, a navigation menu on the left lists various sections: Home, Instructions, Setup, Brute Force, Command Execution, CSRF, File Inclusion, SQL Injection (highlighted in green), SQL Injection (Blind), Upload, XSS reflected, XSS stored, DVWA Security, PHP Info, and About. The main heading is "Vulnerability: SQL Injection". Below this heading is a form with the label "User ID:" and a text input field, followed by a "Submit" button. Underneath the form, there is a "More info" section with three links: <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>, http://en.wikipedia.org/wiki/SQL_injection, and <http://www.unixwiz.net/techtips/sql-injection.html>. The status bar at the bottom shows "Internet" and "100%" zoom.

LAB: SQL injection

<http://127.0.0.1/dvwa/vulnerabilities/sqli/>

- The UserID field here is vulnerable to SQLI
- Attempt to throw an error
- Hint:

```
SELECT first_name, last_name FROM users WHERE user_id =  
'o'brien'
```

- Check out the SQLI Lab sheet....(on your USB Key)....No peeking.

Defending Against SQL Injection

- Stored procedures provide several benefits:
 - ▶ Allows database permissions to be restricted to only **EXECUTE** on stored procedures (permission inheritance)
 - ▶ Promotes code re-use (less error prone and easier to maintain)
 - ▶ They must not contain dynamic SQL
 - ▶ Caution: Stored Procedures themselves may be injectable!

- Query Parameterization Needed:
 - ▶ When creating SQL
 - ▶ When calling a Stored Procedure
 - ▶ When building a Stored Procedure

Restricting Default Database Permissions

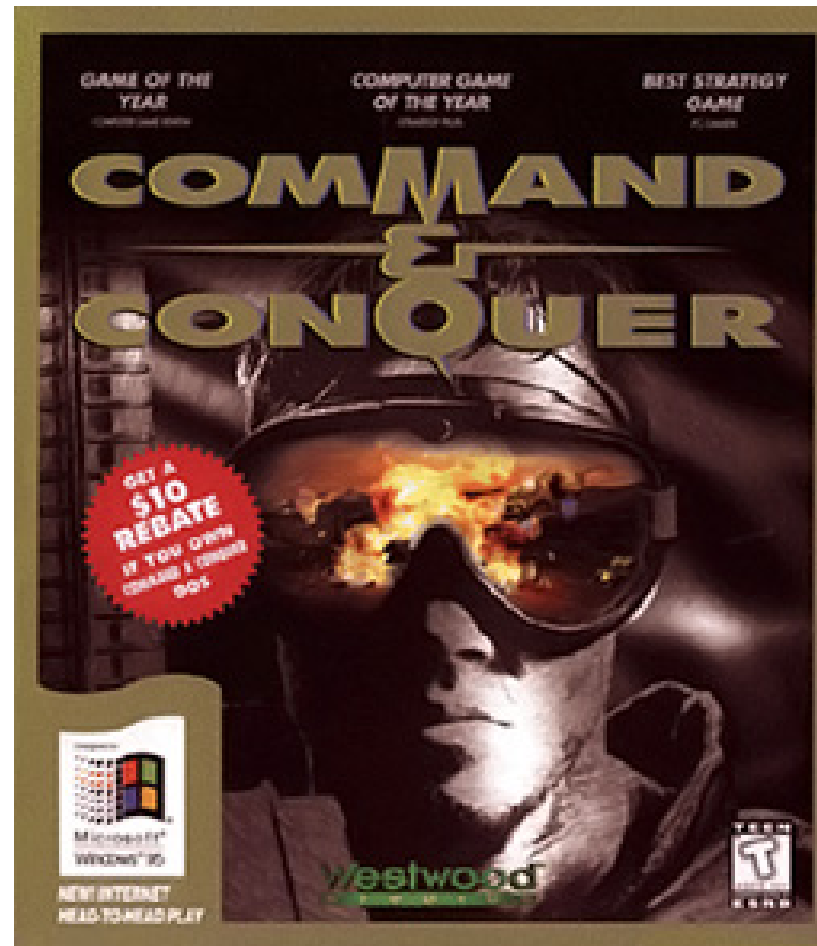
- Delete all default user accounts that are not used. Ensure that strong/complex passwords are assigned to known user accounts.
- Restrict default access permissions on all objects. The application user should either be removed from default roles (i.e. public), or the underlying role permissions should be stripped
- Disable dangerous/unnecessary functionality within the database server (ADHOC provider access and **xp_cmdshell** in Microsoft SQL

Database Principle of Least Privilege

- Database accounts used by the application should have the minimal required privileges.
- If there is a SQLI vuln we may be able to limit the damage that an attacker might do.

DB Query Method	Privileges Required by App	Privileges that can be revoked
Stored Procedure	EXECUTE on the stored procedure	SELECT, INSERT, UPDATE, DELETE on the underlying Tables EXECUTE on system stored procedures SELECT on system tables and views
Dynamic SQL	SELECT on the table (read-only) - OR - SELECT / UPDATE / INSERT / DELETE on the table (read / write)	EXECUTE on system stored procedures SELECT on system tables and views

OS Command Injection



Operating System Interaction

- Applications often pass parameters that are ultimately used to interface with the server file system and/or operating system
- If not validated properly, parameters may be manipulated to provide unauthorized read / write / execute access to server files
- Many applications may allow users to upload files

Arbitrary File Upload

- Uploading malicious files to web-accessible directories can be used to compromise the underlying operating system and/or application
 - ▶ Malicious binaries to executable web-accessible directories (ie. /cgi-bin/)
 - ▶ Malicious scripts to web-accessible directories with script mappings (can be any or all directories)
 - ▶ Overwriting sensitive system files (/etc/passwd, /etc/shadow)
- Uploading large files to the web server can be used to launch a denial-of-service attack by filling web server drives

File Access via Parameters

- Calling other files via input parameters can expose the web server to unauthorized file access

- ▶ /default.jsp?page=about.jsp **OK**

- ▶ /default.jsp?page=../../../../etc/passwd **NOT OK**

Compound this issue with excessive app permissions:

- ▶ /default.jsp?page=../../../../etc/shadow **OH NO!!!**

Injection Flaws -Example

Document retrieval

```
sDoc = Request.QueryString("Doc") ← Source
if sDoc <> "" then
  x = inStr(1,sDoc, ".")
  if x <> 0 then
    sExtension = mid(sDoc,x+1)
    sMimeType = getMime(sExtension)
  else
    sMimeType = "text/plain"
  end if

  set cm = session("cm")
  cm.returnBinaryContent application("DOCUMENTROOT") & sDoc,
  sMimeType
  Response.End
end if
```

Sink
↓

Command Injection

- Web applications may use input parameters as arguments for OS scripts or executables
- Almost every application platform provides a mechanism to execute local operating system commands from application code

- ▶ *Perl: system(), exec(), backquotes(` `)*
- ▶ *C/C++: system(), popen(), backquotes(` `)*
- ▶ *ASP: wscript.shell*
- ▶ *Java: getRuntime.exec*
- ▶ *MS-SQL Server: master..xp_cmdshell*
- ▶ *PHP : include() require(), eval() ,shell_exec*

- Most operating systems support multiple commands to be executed from the same command line. Multiple commands are typically separated with the pipe “|” or ampersand “&” characters

Testing for OS Interaction

- Note any parameters that appear to be referencing files or directory paths. Also note any web server file names or paths that incorporate user specified data

- Parameters should be tested individually to see if file system related errors appear
 - ▶ *File not found, Cannot open file, Path not found, etc.*

- Input parameters should be manipulated to include references to other known files and directories
 - ▶ `../../../../etc/passwd`
 - ▶ `../../../../winnt/win.ini`
 - ▶ `../../../../winnt/system32/cmd.exe`

Testing for OS Interaction

- If the application allows file upload, try and determine where the files are sent. If sent to web accessible directories, upload malicious files and/or script and see if they can be executed
- If not, try to determine the local path to the web root directory and traverse into the directory by manipulating the file name
 - ▶ ../../../../home/apache/htdocs/test.txt
 - ▶ ..\..\..\inetpub\wwwroot\test.txt
- Try appending operating system commands to the end of application parameters. Remember to encode the “&”

Lab - OS file system interaction

<http://127.0.0.1/dvwa/vulnerabilities/exec/>



Home

Instructions

Setup

Brute Force

Command Execution

CSRF

File Inclusion

SQL Injection

SQL Injection (Blind)

Vulnerability: Command Execution

Ping for FREE

Enter an IP address below:

More info

<http://www.scribd.com/doc/2530476/Php-Endangers-Remote-Code-Execution>

<http://www.ss64.com/bash/>

<http://www.ss64.com/nt/>

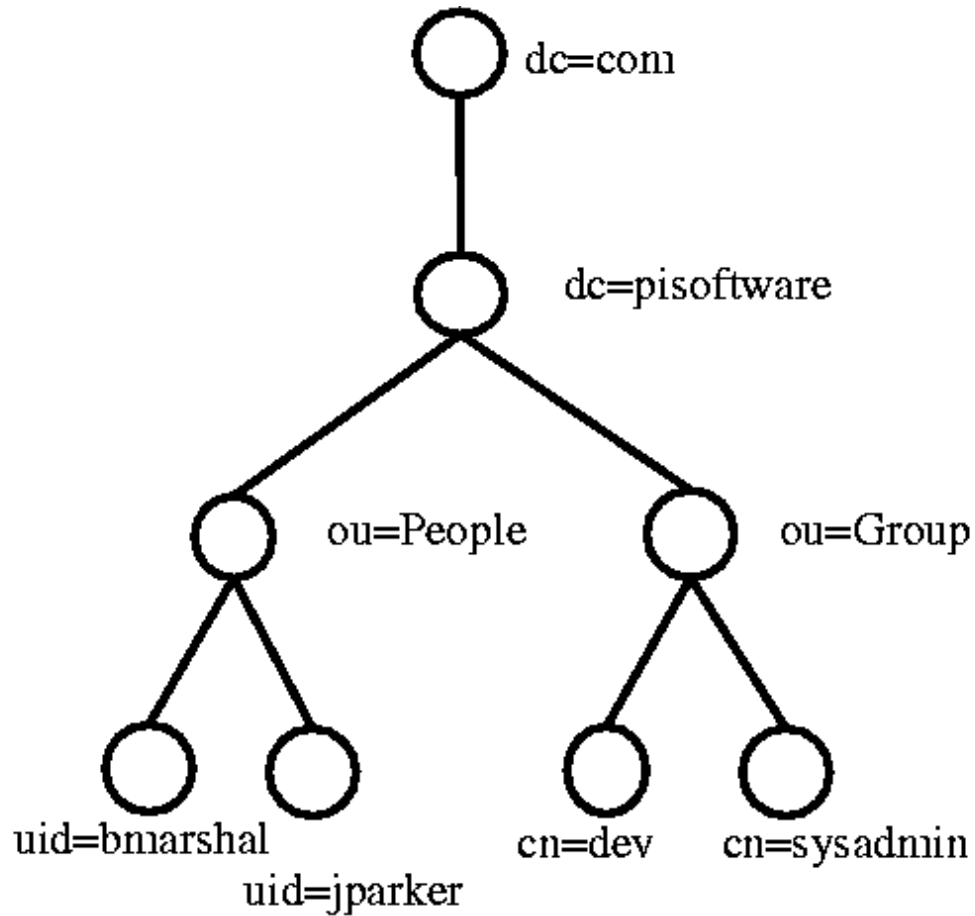
Defenses Against OS Interaction Attacks

- Exact Match Validation should be used to ensure that only authorised files are requested. If this is not feasible, then Known Good Validation or Known Bad Validation should be used on parameter values and characters typically used to alter file system paths should be rejected. (`..` / `%`)
- Bounds Checking should also be performed to ensure that uploaded file sizes do not exceed reasonable limits
- In general, avoid using parameters to interface with the file system when at all possible
- Uploaded files should be placed into a directory that is not web accessible and the application should handle all file naming (regardless of what the original file name was)

Defenses Against OS Interaction Attacks

- For file access using application parameters, consider using application logic to correlate parameter values to file system paths or objects if dynamic file access necessary. This can typically be done using an array or hash table.
- Always implement conservative read, write, and execute access control lists at the OS level to restrict what files can be accessed by the application. (more on this later)
- If possible, verify uploaded file types by inspecting file headers. Native controls for validating file types are available in certain development platforms (.NET)
- Store in application constants, where possible

LDAP Injection



LDAP injection

- Lightweight Directory Access Protocol
- Used for accessing information directories
- Frequently used in web apps to help users search for specific information on the internet.
- Also used for authentication systems.

LDAP Injection

- Technique for exploiting web apps using LDAP statements without first properly validating that data
- Similar techniques involved in SQL injection also apply to LDAP injection
- Could result in the execution of arbitrary commands such as granting permissions to unauthorized queries or content modification inside the LDAP tree
- Can determine how queries are structured by sending logical operators (e.g. OR, AND, |, &, %26) and seeing what errors are returned

LDAP Injection Example

- The following code is responsible to catch input value and generate a LDAP query that will be used in LDAP database:

```
<input type="text" size=20 name="userName">Insert the username</input>
```

- Underlying code for the LDAP query:

```
String ldapSearchQuery = "(cn=" + $userName + ")";  
System.out.println(ldapSearchQuery);
```

- Variable \$username is not validated
- Entering "*" may return all usernames in the directory
- Entering "eoin) (| (password = *))" will generate the following code and reveal eoinpassword:

```
( cn = eoin) ( | (password = * ) )
```

Defenses Against LDAP Injection

- Data input validation of all client-supplied data!
- Use known good validation with a regular expression
 - ▶ Only allow letters and numbers (or just numbers)
 - ▶ `^[0-9a-zA-Z]*$`
- If other characters are needed, convert them to HTML substitutes ("e, >)
- Outgoing data validation
- Access control to the data in the LDAP directory

OWASP Injection Resources

■ LDAP Injection

- ▶ https://www.owasp.org/index.php/LDAP_injection
- ▶ [https://www.owasp.org/index.php/Testing_for_LDAP_Injection_\(OWASP-DV-006\)](https://www.owasp.org/index.php/Testing_for_LDAP_Injection_(OWASP-DV-006))

■ SQL Injection

- ▶ https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
- ▶ https://www.owasp.org/index.php/Abridged_SQL_Injection_Prevention_Cheat_Sheet

■ Command Injection

- ▶ https://www.owasp.org/index.php/Command_Injection