**OWASP**

# SQL INJECTION

## DEFINITION

**Abstract:**

The file act\_save\_direct\_auto\_retail.cfm invokes a SQL query built using input coming from an untrusted source on line 773. This could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

**Explanation:**

SQL injection errors occur when:

1. Data enters a program from an untrusted source. In this case the data enters at Form() in act\_saveverifcollateral.cfm at line 38.

2. The data used to dynamically construct a SQL query.

In this case the data is passed to cfquery() in act\_save\_direct\_auto\_retail.cfm at line 773.

**Example 1:** The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where owner matches the user name of the currently-authenticated user.

<cfquery name="matchingItems" datasource="cfsnippets">

SELECT \* FROM items

WHERE owner='#Form.userName#'

AND itemId=#Form.ID#

</cfquery>

The query that this code intends to execute follows:

**SELECT \* FROM items**

**WHERE owner = <userName>**

**AND itemId = <ID>;**

However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if Form.

ID does not contain a single-quote character.

If an attacker with the user name wiley enters the **string "name' OR 'a'='a"** for Form.ID, then the query becomes the following:

SELECT \* FROM items

WHERE owner = 'wiley'

AND itemId = 'name' **OR 'a'='a';**

**The addition of the OR 'a'='a'** condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

SELECT \* FROM items;

**This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the items table**, regardless of their specified owner.

**Example 2:** This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1.

If an attacker with the user name hacker enters the string **"hacker'); DELETE FROM items; --" for Form.ID**,

then the query becomes the following two queries:

SELECT \* FROM items

WHERE owner = 'hacker'

AND itemId = **'name';**

**DELETE FROM items;**

**--'**

**Many database servers, including Microsoft(R) SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once**.

**While this attack string results in an error on Oracle** and other database servers that do not allow the batch-execution of statements separated by semicolons, on databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed [4].

In this case the comment character serves to remove the trailing single-quote left over from the modified query. On a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1.

If an attacker enters the string "name'); DELETE FROM items; SELECT \* FROM items WHERE 'a'='a", the following three valid statements will be created:

SELECT \* FROM items

WHERE owner = 'hacker'

AND itemId = 'name';

**DELETE FROM items;**

**SELECT \* FROM items WHERE 'a'='a';**

## SOLUTIONS

**One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem** and

**either accept only characters from a whitelist of safe values or identify and escape a blacklist of potentially malicious values**.

Whitelisting can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security.

As is almost always the case, blacklisting is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers may:

- Target fields that are not quoted

- Find ways to bypass the need for certain escaped meta-characters

- Use stored procedures to hide the injected meta-characters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures.

Although stored procedures prevent some types of SQL injection attacks, they fail to protect against many others.

Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters.

However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures.

Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.

## RECCOMENDATIONS

The root cause of a SQL injection vulnerability is the ability of an attacker to change context in the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead.

When a SQL query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data.

**Parameterized SQL statements** can enforce this behavior by disallowing data-directed context changes and preventing **nearly all SQL injection attacks**.

Parameterized SQL statements are constructed using strings of regular SQL, but when user-supplied data needs to be included, they create bind parameters, which are placeholders for data that is subsequently inserted.

**Bind parameters allow the program to explicitly specify to the database what should be treated as a command and what should be treated as data**.

When the program is ready to execute a statement, it specifies to the database the runtime values to use for the value of each of the bind parameters, without the risk of the data being interpreted as commands.

The previous example can be rewritten to use parameterized SQL statements (instead of concatenating user supplied strings) as follows:

<cfquery name="matchingItems" datasource="cfsnippets">

SELECT \* FROM items

FROM items

WHERE owner = <cfqueryparam value="#Form.userName#"

cfsqltype="cf\_sql\_varchar">

AND itemname = <cfqueryparam value="#Form.ID#"

cfsqltype="cf\_sql\_varchar">

</cfquery>

More complicated scenarios, often found in report generation code, require that user input affect the command structure of the SQL statement, such as the addition of dynamic constraints in the WHERE clause.

Do not use this requirement to justify concatenating user input into query strings.

Prevent SQL injection attacks where user input must affect statement command structure with a level of indirection: create a set of legitimate strings that correspond to different elements you might include in a SQL statement. When constructing a statement, use input from the user to select from this set of application-controlled values.

**Use bind parameters whenever input data must be directly included in a statement.**

**Tips:**

1. A common mistake is to use parameterized SQL statements that are constructed by concatenating user-controlled strings. Of course, this defeats the purpose of using parameterized SQL statements. If you are not certain that the strings used to form statements are constants controlled by the application, do not assume that they are safe because they are not being executed directly as SQL strings. Thoroughly investigate all uses of user-controlled strings in SQL statements and verify that none can be used to modify the meaning of the query.

# Ldap injection

Username sanitize to prevent LDAP injection

|  |
| --- |
| // Sanitize username to prevent against LDAP Injection. We will throw exception  // if the username contains a wildcard (\*) or if it matches the following pattern:  // [starts with 0 or more characters][closing paren][0 or more characters][opening paren][1 or more characters][! or = or ! or > or <][0 or more characters at end]  // The characters can be any characters (alphanumberic and symbols)  **if**(user.matches("^.\*\\).\*\\(.+[!=~><].\*$") || user.contains("\*")) {  log(0, "Authentication Exception. Possible LDAP Injection attack with username: " + user);  **throw** **new** AuthenticationException("Authentication Exception. Possible LDAP Injection attack with username: " + user);  } |

Also Occurs when call is made to SEARCH() method

Example and solutions

|  |
| --- |
| import java.sql.Connection;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.util.ArrayList;  import java.util.HashMap;  import java.util.Hashtable;  import java.util.Map;  import javax.naming.AuthenticationException;  import javax.naming.Context;  import javax.naming.NamingEnumeration;  import javax.naming.NamingException;  import javax.naming.directory.Attribute;  import javax.naming.directory.Attributes;  import javax.naming.directory.SearchControls;  import javax.naming.directory.SearchResult;  import javax.naming.ldap.InitialLdapContext;  import javax.naming.ldap.LdapContext;  /\*\*  \* This class tests LDAP authentication in Java  \*/  public class ADAuthenticator {  private static String prev\_iniFile = "";  private String domain;  private String ldapHost;  private String searchBase;    private static int debug\_level = 0;  private static String logFile = "";  private static String clientName = "";  private static int prev\_debug\_level = 0;  private static String prev\_logFile = "";  private static String prev\_clientName = "";    private ArrayList<String> ldapFilterName = new ArrayList<String>();  private ArrayList<String> ldapFilterValue = new ArrayList<String>();  private String evaluatorId = "";    private LogMsg log\_obj = new LogMsg();    private Map attributeMap = null;  public ADAuthenticator(String evalId, String iniFile) {  this.evaluatorId = evalId;    readINI(iniFile,null);  }    //new constructor added for 165669  public ADAuthenticator(String evalId, IniFile ini\_in) throws Exception {  this.evaluatorId = evalId;  if(ini\_in != null){  readINI(null,ini\_in);  }  else {  throw new Exception("Null value for ini object.");  }  }    private void readINI(String iniFile, IniFile ini\_in) {  try {  if(ini\_in == null && iniFile != null){  // Don't read INI again if already read before  if(!prev\_iniFile.equalsIgnoreCase(iniFile)) {  IniFile ini = new IniFile();  ini.readINIFile(iniFile);  debug\_level = Integer.valueOf(ini.getINIVar("debug.debug\_level", "0"));  logFile = ini.getINIVar(LoggingConstants.DEBUG\_LOG,"");  clientName = ini.getINIVar("general.client\_name");  prev\_debug\_level = debug\_level;  prev\_logFile = logFile;  prev\_clientName = clientName;  prev\_iniFile = iniFile;  } else {  clientName = prev\_clientName;  debug\_level = prev\_debug\_level;  logFile = prev\_logFile;  }  } else {  IniFile ini = ini\_in;  debug\_level = Integer.valueOf(ini.getINIVar("debug.debug\_level", "0"));  logFile = ini.getINIVar(LoggingConstants.DEBUG\_LOG,"");  clientName = ini.getINIVar("general.client\_name");    }    if(logFile != null && logFile.length() > 0) {  log\_obj.openLogFile(logFile);  }    javax.naming.InitialContext ctx = null;  Connection con = null;  try {  ctx = new javax.naming.InitialContext();  javax.sql.DataSource ds = (javax.sql.DataSource)ctx.lookup(clientName+"\_origpool");  con = ds.getConnection();  if(con==null) {  throw new Exception("Connection is null");  }  } catch (Exception e) {  throw new Exception("Exception initializing db connection: " + e.toString());  }    // Get the LDAP configuration  String select = "select ldap\_host\_txt, ldap\_domain\_txt, ldap\_search\_base\_txt " +  "from config\_ldap " +  "where evaluator\_id = ? ";  String selectFilters = "select attribute\_name\_txt, attribute\_value\_txt " +  "from config\_ldap\_filters " +  "where evaluator\_id = ? ";  PreparedStatement ps = null;  ResultSet rs = null;  try {  ps = con.prepareStatement(select);  ps.setInt(1, Integer.valueOf(evaluatorId));  rs = ps.executeQuery();  if(rs.next()) {  domain = rs.getString("ldap\_domain\_txt");  ldapHost = rs.getString("ldap\_host\_txt");  searchBase = rs.getString("ldap\_search\_base\_txt");    rs.close();  ps.close();    ps = con.prepareStatement(selectFilters);  ps.setInt(1, Integer.valueOf(evaluatorId));  rs = ps.executeQuery();  while(rs.next()) {  ldapFilterName.add(rs.getString("attribute\_name\_txt"));  ldapFilterValue.add(rs.getString("attribute\_value\_txt"));  }  } else {  throw new Exception("No LDAP configuration found for evaluator\_id "+evaluatorId+".");  }  } catch (Exception e) {  throw e;  } finally {  /\*\*  \* OWASP TOP 10 2013 - A1 NULL Dereference  \* TTP 346385 Security Remediation Fortify Scan  \*\*/  try{if(rs != null)rs.close();}catch(Exception e){}  try{if(ps != null)ps.close();}catch(Exception e){}  try{con.close();}catch(Exception e){}  }    } catch (Exception e) {  debug\_level = 0;  log(0, "Exception reading Origenate INI: ", e);  }  }    private void log(int level, String msg) {  log\_obj.FmtAndLogMsg(msg, debug\_level, level);  }    private void log(int level, String msg, Throwable throwable) {  log\_obj.FmtAndLogMsg(msg, throwable, debug\_level, level);  }    public String getAttributeValue(String att) {  if(att == null || att.equals("")) return "";    if(attributeMap.containsKey(att))  return (String)attributeMap.get(att);  else  return "";  }    public boolean authenticateUser(String user, String pass) {  try {  attributeMap = authenticate(user, pass);  if(attributeMap == null) {  throw new Exception("User authenticated, but not validated in LDAP.");  }  log(0,"Successfully authenticated");  log(4, "Attributes: " + attributeMap.toString());  return true;  } catch (Exception e) {  log(0, "Exception authenticating user " + user + ": ", e);  e.printStackTrace();  }  return false;  }  public Map authenticate(String user, String pass) throws NamingException,  AuthenticationException, Exception {  // Sanitize username to prevent against LDAP Injection. We will throw exception  // if the username contains a wildcard (\*) or if it matches the following pattern:  // [starts with 0 or more characters][closing paren][0 or more characters][opening paren][1 or more characters][! or = or ! or > or <][0 or more characters at end]  // The characters can be any characters (alphanumberic and symbols)  if(user.matches("^.\*\\).\*\\(.+[!=~><].\*$") || user.contains("\*")) {  log(0, "Authentication Exception. Possible LDAP Injection attack with username: " + user);  throw new AuthenticationException("Authentication Exception. Possible LDAP Injection attack with username: " + user);  }    StringBuilder temp\_builder = new StringBuilder("");  temp\_builder.append("(&(objectClass=user)(sAMAccountName=");  temp\_builder.append(user);  temp\_builder.append(")");    int filterLength = ldapFilterName.size();  for(int i = 0; i < filterLength; i++) {  String filterName = ldapFilterName.get(i);  String filterValue = ldapFilterValue.get(i);  if(filterName != null && filterValue != null && !filterName.equals("") && !filterValue.equals("")) {  if(filterName.matches("^.\*\\).\*\\(.\*$") || filterName.contains("\*")) {  log(0, "Authentication Exception. Possible LDAP Injection attack with filter name: " + filterName);  throw new AuthenticationException("Authentication Exception. Possible LDAP Injection attack with filter name: "+filterName);  }    if(filterValue.matches("^.\*\\).\*\\(.+[!=~><].\*$") || filterValue.contains("\*")) {  log(0, "Authentication Exception. Possible LDAP Injection attack with filter value: " + filterValue);  throw new AuthenticationException("Authentication Exception. Possible LDAP Injection attack with filter value: "+filterValue);  }  temp\_builder.append("("+filterName+"="+filterValue+")");  }  }  temp\_builder.append(")");  String searchFilter = temp\_builder.toString();  log(4, "LDAP Search Filter: " + searchFilter);  // Create the search controls  SearchControls searchCtls = new SearchControls();  // Specify the search scope  searchCtls.setSearchScope(SearchControls.SUBTREE\_SCOPE);  Hashtable env = new Hashtable();  env.put(Context.INITIAL\_CONTEXT\_FACTORY,  "com.sun.jndi.ldap.LdapCtxFactory");  env.put(Context.PROVIDER\_URL, ldapHost.toLowerCase());  env.put(Context.SECURITY\_AUTHENTICATION, "simple");  env.put(Context.SECURITY\_PRINCIPAL, user + "@" + domain.toUpperCase());  env.put(Context.SECURITY\_CREDENTIALS, pass);    log(4, "Attempting to authenticate with LDAP Host: " + ldapHost.toLowerCase() +  ", User Principal: " + user + "@" + domain.toUpperCase());  LdapContext ctxGC = null;  ctxGC = new InitialLdapContext(env, null);  /\*\*Search objects in GC using filters  NamingEnumeration answer = ctxGC.search(searchBase, searchFilter,  searchCtls);\*/    /\*\*  \* OWASP TOP 10 - 2010 LDAP Injection  \* Changes to the below code to fix Vulnerabilites  \* False Positive  \*/  // fortify false positive  NamingEnumeration answer = ctxGC.search(searchBase, searchFilter,  searchCtls);    while (answer.hasMoreElements()) {  SearchResult sr = (SearchResult) answer.next();  Attributes attrs = sr.getAttributes();  Map amap = null;  if (attrs != null) {  amap = new HashMap();  NamingEnumeration ne = attrs.getAll();  while (ne.hasMore()) {  Attribute attr = (Attribute) ne.next();  amap.put(attr.getID(), attr.get());  }  ne.close();  }  ctxGC.close();  return amap;  }  ctxGC.close();    return null;  }  } |

# Dynamic code evaluation in Java script

Mostly with settimeout() function, when you pass a string to it.

It can lead to code injection

Example

|  |
| --- |
| str="callAjax('divQueueFilter','<%=request.getContextPath()%>/getAppUpdateQueueFilter.do?originator="+originator+"&app\_id="+app\_id+"&update\_field="+update\_field+"&update\_value="+update\_value+"&remain\_apps="+remain\_apps+"&selected\_apps="+selected\_apps+"&sales\_ref\_id\_bfr\_save="+sales\_ref\_id\_bfr\_save+"&mgr\_id\_bfr\_save="+mgr\_id\_bfr\_save+"&app\_status\_bfr\_save="+app\_status\_bfr\_save+"&secs="+seed.valueOf()+"',false,false,"+disable\_button\_flg+")";  setTimeout(str, 10); |

Fix: Never pass a string PARAM to setTimeout() function

|  |
| --- |
| var Uri='<%=request.getContextPath()%>/getAppUpdateQueueFilter.do?originator="+originator+"&app\_id="+app\_id+"&update\_field="+update\_field+"&update\_value="+update\_value+"&remain\_apps="+remain\_apps+"&selected\_apps="+selected\_apps+"&sales\_ref\_id\_bfr\_save="+sales\_ref\_id\_bfr\_save+"&mgr\_id\_bfr\_save="+mgr\_id\_bfr\_save+"&app\_status\_bfr\_save="+app\_status\_bfr\_save+"&secs="+seed.valueOf()+"';  uri=encodeURIComponent(uri);  setTimeout( callAjax('divQueueFilter',uri, false, false, disable\_button\_flg),10); |

# XPATH INJECTION IN XML

Sol: USE xpath api from com.sun.org.apache.xpath.internal.XPathAPI

|  |
| --- |
| XPathAPI.selectSingleNode(doc,xpath+"/AreaCode/text()"); |

# FILE PATH MANIPULATION

Directory path manipulation

File name path manipulation

|  |
| --- |
| **public** **static** **final** String ***REGEXVALIDATEPATH*** = "^([a-zA-Z]:)?([\\\\|/]\*[a-zA-Z0-9':,= \_\\$\\&\\.\\+\\-\\(\\)]+)+[\\\\|/]?$";  using this regex  if(input.matches(VALIDATEPATH)){  return input;  } |

# JSON Server Side INJECTION

Use GOOGLE’s json-sanitizer jar

# Command Injection

|  |
| --- |
| // To avoid Command Injection, need to scrub out all of the following:  // ; | || && < > >>  String strArg = s\_name[j].replaceAll("[;|><]+", "").replace("&&", ""); |

# Header Injection

## CR LF injection

Attack value that can send

|  |
| --- |
| String contentType="text/html%0d%0a%0d%0a%3Cscript%3Ealert(1)%3C/script%3E"; |

Solution

Sanitize header data for Cr LF

You should use a wrapper filter for servlet but you can also sanitize it at class level

Wrapper level code

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | public class CRLFPreventionFilter implements Filter { | |  | private static final String CRLF\_CONFIG\_ENABLED\_PROPERTY = "Security.CRLFPreventionConfig.Enabled"; | |  | private static boolean CRLFPreventionEnabled = false; | |  |  | |  | @Override | |  | public void init(FilterConfig filterConfig) throws ServletException { | |  | // Loads enabled configuration at /repository/conf/carbon.xml//Server/Security/CRLFPreventionConfig/Enabled | |  | ServerConfiguration serverConfiguration = ServerConfiguration.getInstance(); | |  | if (serverConfiguration.getFirstProperty(CRLF\_CONFIG\_ENABLED\_PROPERTY) != null && Boolean.parseBoolean( | |  | serverConfiguration.getFirstProperty(CRLF\_CONFIG\_ENABLED\_PROPERTY))) { | |  | CRLFPreventionEnabled = true; | |  | } | |  | } | |  |  | |  | @Override | |  | public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, | |  | FilterChain filterChain) throws IOException, ServletException { | |  |  | |  | if (CRLFPreventionEnabled && servletResponse instanceof HttpServletResponse) { | |  | CRLFResponseWrapper responseWrapper = new CRLFResponseWrapper((HttpServletResponse) servletResponse); | |  | filterChain.doFilter(servletRequest, responseWrapper); | |  | } else { | |  | filterChain.doFilter(servletRequest, servletResponse); | |  | } | |  | } | |  |  | |  | @Override | |  | public void destroy() { | |  | // Nothing to implement | |  | } | |  |  | |  | protected static class CRLFResponseWrapper extends HttpServletResponseWrapper { | |  |  | |  | public CRLFResponseWrapper(HttpServletResponse response) { | |  | super(response); | |  | } | |  |  | |  | @Override | |  | public void addCookie(Cookie cookie) { | |  | cookie.setValue(sanitize(cookie.getValue())); | |  | super.addCookie(cookie); | |  | } | |  |  | |  | @Override | |  | public void addHeader(String name, String value) { | |  | super.addHeader(sanitize(name), sanitize(value)); | |  | } | |  |  | |  | @Override | |  | public void setHeader(String name, String value) { | |  | super.setHeader(sanitize(name), sanitize(value)); | |  | } | |  |  | |  | @Override | |  | public void sendRedirect(String location) throws IOException { | |  | super.sendRedirect(sanitize(location)); | |  | } | |  |  | |  | private String sanitize(String input) { | |  |  | |  | if (StringUtils.isBlank(input)) { | |  | return input; | |  | } | |  |  | |  | return input.replaceAll("(\\r|\\n|%0D|%0A|%0a|%0d)", ""); | |  | } | |  | } | |  |  | |  | } | |

Class level code

|  |
| --- |
| **public** String SanitizeForCRLF(String input){  **if**(!input.isEmpty()){  **return** input.replaceAll("(\\r|\\n|%0D|%0A|%0a|%0d)", "");    }  } |

# Mass Assignment vulnerability Spring

https://www.owasp.org/index.php/Mass\_Assignment\_Cheat\_Sheet