PARASOFT®

Preventing Security Problems with AEP

Security is a complex and often overwhelming issue. To ensure application security, you must not only prevent hackers from entering the system, but also ensure that code safeguards security if a hacker does break in. Moreover, there is no room for error. If you anticipate and prevent hundreds of security vulnerabilities but overlook just one, a hacker can still wreak havoc on your system.

Three of the most commonly exploited internal software weaknesses are:

- Dangerously-constructed SQL statements (for programs that interact with a database).
- Buffer overflows (for C and C++).
- Uncaught runtime exceptions (for Java as well as .NET-based languages such as managed C and C++).

The traditional industry approach to avoiding these attacks is to build code, then later perform a sort of "monkey testing" intended to simulate hacker actions. Testers attempt to design and execute a large number and variety of tests which pound on the application in as many different ways as possible— all in hopes that these tests will reveal a security vulnerability, which can then be remedied prior to deployment. If applied thoroughly, this strategy can expose many of the critical security vulnerabilities in the code. However, thorough application of this strategy is difficult and time-consuming. For this type of testing to be effective, the tester must think like a hacker, and break the program's normal defenses in order to reach the root of the problem, and then identify the application's security vulnerabilities. Moreover, the test cases that must be created to identify these security vulnerabilities are typically complex, and few teams have the time and resources to write the necessary number and range of complex test cases without slipping on their deadlines and budget.

An easier way to protect code from these three common attacks is to supplement security testing with a concerted effort to prevent security vulnerabilities from being introduced as the team is writing code. The Parasoft Automated Error Prevention (AEP) Methodology provides an effective and feasible way to prevent security vulnerabilities through the automated application of industry-standard best practices, such as static analysis, dynamic analysis, unit testing, and runtime error detection. When you apply these preventative practices, you can start improving your code's security before you write test cases specifically for security verification; this allows you to jumpstart your security efforts and provides a practical way to improve application security. By applying the recommended AEP practices throughout the software development lifecycle, it's possible to remove many security vulnerabilities, as well as improve the overall code quality and reliability.

This article discusses how hackers exploit dangerously-constructed SQL statements, buffer overflows, and uncaught runtime exceptions, then explains how AEP helps remove these vulnerabilities, essentially stopping hackers in their tracks.

SQL Injection

When SQL statements are dynamically created as software executes, there is an opportunity for a security breach: if the hacker is able to pass fixed inputs into the SQL statement, then these inputs can become part of the SQL statement. If the hacker knows his SQL, he can use this technique to gain access to privileged data, login to password-protected areas without a proper login, remove database tables, add new entries to the database, or even login to an application with admin privileges.

SQL Injection Examples

The following example shows a system that was designed to restrict site access to only registered users. Unfortunately, a hacker can use SQL injection to do much more.

```
Http login form:

<FORM name=login action=login.jsp METHOD=post>
User name: <input name="USER">
<br>
Password: <input type="password" name="PASSWORD">
<br>
<input type="submit" value="Go">
</FORM>


login.jsp:

<%@ page import="java.sql.*" %>
<%@ page import="java.io.*" %>

<%
    //MySQL
    String DRIVER = "org.gjt.mm.mysql.Driver";
    String DBURL = "jdbc:mysql://localhost:3306/fruits";
    String LOGIN = "fruits";
    String PASSWORD = "fruits";

    Class.forName(DRIVER);
    Connection connection = DriverManager.getConnection(DBURL, LOGIN, PASSWORD);

    String sUsername = request.getParameter("USER");
    String sPassword = request.getParameter("PASSWORD");
    int iUserID = -1;
    String sLoggedUser = "";

    String s = "SELECT User_id, Username FROM USERS WHERE Username = '" +
                sUsername + "' AND Password = '" + sPassword + "'";
    Statement selectStatement = connection.createStatement ();
    ResultSet resultSet = selectStatement.executeQuery(s);
    if (resultSet.next()) {
        iUserID = resultSet.getInt(1);
        sLoggedUser = resultSet.getString(2);
    }
```

```
    PrintWriter writer = response.getWriter ();
    if (iUserID >= 0) {
        writer.println ("User logged in: " + sLoggedUser);
    } else {
        writer.println ("Access Denied!");
    }
%>
```

The database table that stores user accounts looks like this:

```
+---------+-------------+----------+
| User_id | Username    | Password |
+---------+-------------+----------+
|       1 | john        | doe      |
|       2 | jan's friend | friend   |
|       3 | admin       | qaz456   |
+---------+-------------+----------+
```

The intended usage is that when a user provides the inputs `user=john` and `password=doe`, the string `select User_id, Username from users where Username='john' and Password='doe'` is formed and the user will be logged in as John.

However, there are many ways that a hacker can use SQL injection to perform actions that the developer did not anticipate.

Logging in Without a Valid Username and Password

First, imagine that the hacker submits the following inputs:

```
user = ' or 1=1 #
password = (ANY)
```

The resulting SQL statement would be `select User_id, Username from users where Username='' or 1=1 #' and Password='(ANY)'`. As a result, the hacker can log in as John without providing a valid username or password.

Note that we are using '#' because this example connects to a MySQL database, which takes '#' as a comment. For other databases, other comment delimiters may apply (for example, '--').

Logging in as the Administrator

The information exposed by error messages can be very useful to a hacker. Imagine that the hacker submits the following inputs:

```
user = ' error!
password = (ANY)
```

The resulting SQL statement would be `select User_id, Username from users where Username='' error! ' and Password='(ANY)'`. As a result, the hacker receives a valuable error message:

```
java.sql.SQLException: Syntax error or access violation: You have an error
in your SQL syntax near 'error! ' AND Password = ''' at line 1
```

Now, the hacker knows that the column names are based on their functions (for example, that a column with passwords is named "password"), so he can make educated guesses about the column names. For instance, he might try to submit the following inputs:

```
user = ' or Username like '%' or User_name like '%' #
password = (ANY)
```

As a result, he will receive another error message:

```
java.sql.SQLException: Column not found: Unknown column 'User_name' in
'where clause'
```

This response tells the hacker that `Username` was a good guess. He then submits another set of inputs:

```
user = ' or Username like 'a%' #
password = (ANY)
```

The resulting SQL statement would be `select User_id, Username from users where Username='' or Username like 'a%' # ' and Password='(ANY)'`. As a result, the hacker would be logged in as admin, and gain all associated privileges.

Deleting Tables

Even without logging in as admin, the hacker can remove database tables. Assume that the hacker provides the following inputs:

```
user = ' or 1=1; drop table users; #
password = (ANY)
```

The resulting SQL statement would be `select User_id, Username from users where Username='' or 1=1; drop table users; # ' and Password='(ANY)'`. As a result, the hacker would be logged in as John, and the `users` table would be deleted if the database engine and driver allow multiple SQL statements to be passed as one. Fortunately, most JDBC drivers do not.

SQL Injection Prevention

The traditional attempt to avoid this problem is to validate all user inputs. This is generally an effective way of dealing with malicious user input. However, it's possible to prevent these attacks

altogether by building the statements in such a way that it is impossible for hackers to hijack them—even with the most well-designed and malicious inputs.

A simple way to ward off SQL injection attacks is to ensure that all SQL statements recognize user inputs as variables, and that statements are precompiled before the actual inputs are substituted for the variables. Typically, this is implemented as a two-stage process. In the first stage, the SQL statement should be built and parsed with variables in place of the expected user inputs. Then, in the second stage—before the statement is passed to the database— the variables should be replaced with the user inputs. When you implement this strategy, you ensure that user inputs are never parsed as the actual SQL statement, so even malicious user inputs are rendered ineffective.

For instance, in Java, a secure way to build SQL statements is to construct all queries with `PreparedStatement` instead of `Statement` and/or to use parameterized stored procedures. Parameterized stored procedures are compiled before user input is added, making it impossible for a hacker to modify the actual SQL statement. When `PreparedStatement` is used, most JDBC drivers will prepare a statement with the server, and then supply the parameters separately. In either case, after the initial parsing, there is a clear distinction between the SQL statement and the variable. The variables are encapsulated and special characters within them are automatically escaped in a manner suited to the target database. Consequently, it is impossible for a hacker to pass malicious input and have it treated as if it were the actual SQL statement— treatment that is required if the hacker is going to succeed with SQL injection attacks.

Even if you use `PreparedStatement`, you still need to pay attention to the way in which you build arguments. All parameters should be inserted through appropriate JDBC calls. If you concatenate the SQL sentence and omit the JDBC calls, then an attempted SQL injection could be parsed as SQL, and the hacker could succeed.

For example, the following code illustrates both the correct and incorrect ways of using `PreparedStatement`.

```
package fruits;


import java.sql.*;



public class CustomerDatabaseJDBC extends CustomerDatabase
{


    public int getUserId(String user, String password)
    {
        ConnectionManager manager = ConnectionManager.getInstance ();
        Connection connection = manager.getConnection ();


        PreparedStatement selectStatement = null;
        ResultSet resultSet = null;
```

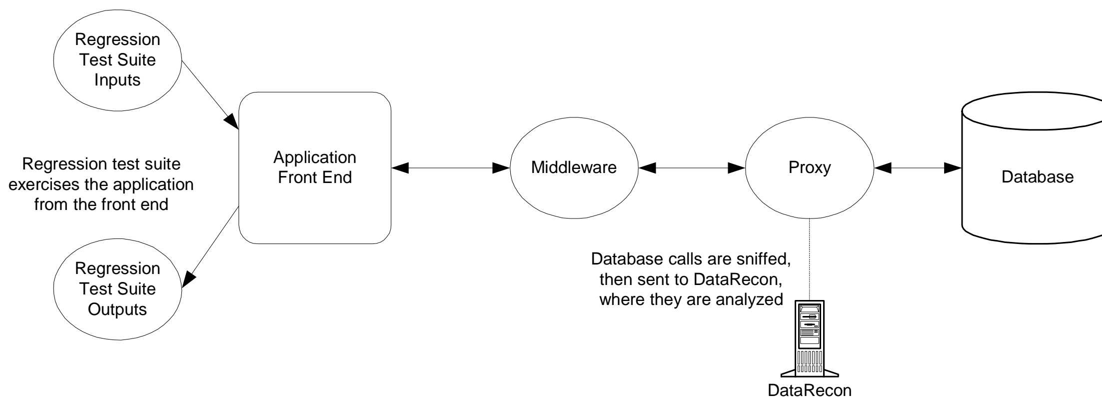
```
        int id = -1;
        try {
            /* Correct use of prepared statement. Both parameters are inserted
               through appropriate JDBC calls
            selectStatement = connection.prepareStatement(
                "SELECT User_id FROM USERS WHERE Username = ? AND Password = ?");


            selectStatement.setString (1, user);
            selectStatement.setString (2, password);
            */


            /* Incorrect use of prepared statement. Developer is concatenating
             SQL sentence and omits JDBC calls */
            selectStatement = connection.prepareStatement(
                "SELECT User_id FROM USERS WHERE Username = '" +
                user +
                "' AND Password = '" +
                password + "'");


            /* Still incorrect use of prepared statement.
               All parameters should be passed through JDBC calls
            selectStatement = connection.prepareStatement(
                "SELECT User_id FROM USERS WHERE Username = '" +
               user + "' AND Password = ?");


            selectStatement.setString (1, password);
            */


            resultSet = selectStatement.executeQuery();


            if (resultSet.next()) {
                id = resultSet.getInt(1);
            }
        } catch (SQLException exception) {
            System.err.println ("Error looking for user: " +
                                exception.getMessage ());
        } finally {
            if (resultSet != null) {
                try {resultSet.close();} catch (SQLException ex) {}
            }
            if (selectStatement != null) {
                try {selectStatement.close();} catch (SQLException ex) {}
            }
            manager.reclaimConnection (connection);
        }



        return id;
    }
}
```

How do you ensure that code follows these best practices? Most SQL statements are created dynamically; consequently, you need to execute the application paths that create

SQL statements and verify whether they are being constructed in a secure manner (i.e., that the statements are precompiled with variables before user input is added). In addition, you need to inspect the database-related code to verify that secure coding practices are being followed (for instance, the Java best practices of using `PreparedStatement` instead of `Statement`, and for using `PreparedStatement` correctly).

AEP solutions for Java, C/C++ and .NET automate both of these types of verification as part of their application of best practices. For instance, in Java, AEP technologies can statically analyze the code that is responsible for forming the SQL statements. This static analysis can be used to verify that if you build SQL for JDBC, you always use `PreparedStatement`. It can also verify whether all available `PreparedStatements` it includes are built properly (with all parameters inserted through appropriate JDBC calls, rather than string concatenation).

Moreover, to determine whether SQL statements are being built in the recommended two-step process, the AEP monitoring technologies use dynamic analysis to watch database calls as the application is being tested during the integration phase. As the application is stimulated through a functional test suite that is run through a test client, the AEP monitoring technology watches the test suite coverage to see which application paths are covered, and traps all SQL statements that pass through the proxy. Parasoft DataRecon, an AEP product for databases, then analyzes how the SQL statements were dynamically constructed, and identifies any statements that were built in an unsafe way. In this way, AEP allows you to identify security vulnerabilities and correct them before hackers have the opportunity to exploit your application. This analysis process is illustrated in the following figure.



While the previous examples refer to Java and JDBC, the same principles can be applied to code built in C++, ASP, Visual Basic, etc. For instance, with C++, you would perform runtime instrumentation to enable the integration testing monitoring, and you would again supplement this dynamic analysis with a static analysis of database-related code. In all cases, the key to security is to ensure that the user input is not recognized and parsed as part of the SQL statement.

Buffer Overflow

Buffer overflows are a standard problem that affect C/C++ applications all too often. Buffer overflow attacks occur when a hacker manages to pass an input through all the program's built-in defenses and write to the buffer. It is only possible when the hacker is able to find and exploit a memory corruption bug in the application.

Buffer Overflow Example

For example, assume that your C++ application has an array or a memory chunk on the stack, and a memory issue makes it possible to write beyond the size of the array or memory chunk, and overwrite the return address of the function. The hacker can exploit this weakness so that the function returns to a hacker-designated function, or so that the function executes a hacker-designated operation. In fact, some recently-discovered buffer overflows in major operating systems provide hackers license to perform a variety of malicious actions, such as causing the system to fail, installing programs, viewing, changing, and deleting data, modifying any part of the system, and creating accounts with full privileges.

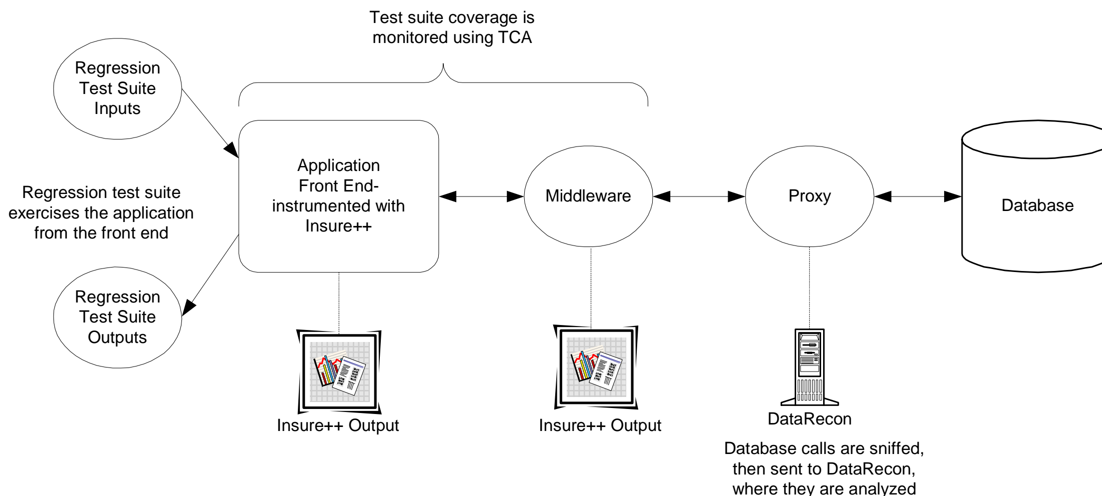
Buffer Overflow Prevention

Developers traditionally try to handle these exploits by limiting the size of the input or by verifying the input. However, it's easy to miss cases if you have no procedure for identifying all the inputs which need to be checked. To actually remove the opportunity for these attacks, you need to prevent the memory corruption bugs that allow them to occur.

Memory corruption bugs can be prevented by applying the standard AEP C/C++ integration testing practices. During integration testing, the application is compiled with the Insure++ runtime error detection tool, which detects memory corruption and pinpoints where the program is overwriting memory. As the application is being stimulated and exercised by the regression test suites, any memory corruption is identified— every chunk of memory is observed and every memory access is checked to verify whether it is legal and to determine whether it can overwrite the range, resulting in a buffer overflow. Identified problems are reported, and should be fixed as soon as possible.

If the test suite thoroughly exercises the application, most possibilities for memory corruption should be exposed by this process. To verify whether the application is being exercised thoroughly, you can monitor coverage during testing to identify unexercised code, then add test cases as needed to achieve fuller cover. For instance, when you are testing code with Insure++, the Parasoft TCA tool can identify which blocks of code were covered and which were not.

This process is illustrated in the following figure:



Uncaught Runtime Exception Vulnerabilities

People believe that languages such as Java and .NET (managed C/C++) don't have a problem with buffer overflows. This is true, but they do suffer from a different problem that is just as serious: uncaught runtime exceptions. Typical checked exceptions provide a relatively easy way to transfer flow and keep a program running when exceptional situations occur. However, unchecked runtime exceptions— exceptions that are automatically thrown by the runtime system when a program violates the language syntax/semantics— are usually an indication of software bugs. They typically stem from problems related to arithmetic, pointers, and indexing, and can occur at any point in a program. If these exceptions surface in the field, the resulting unexpected flow transfer and potential thread termination could lead to instability, unexpected results, or even crashes or security breaches.

Uncaught Runtime Exception Example

For an example of how uncaught runtime exceptions can create a security vulnerability, consider the following sample Java code, where a very simple `NullPointerException` in login code allows a hacker to completely bypass the login procedure:

```
// Data.java

public class Data {
    public String [] getSecretData (String name, String password) throws UnauthenticatedException
{
        if (Login.isValidLogin(name,password)) {
            return secretData;
        } else {
            throw new UnautenticatedException("Not Authorized");
        }
    }
}


// XpresUserFinder.java
public class XpresUserFinder {
    public static boolean isUsernamePasswordCorrect(
```

```
        String s_name,
        String s_password)
        throws DataException {
        //
        IXpresUser ixuse = null;
        //
        ixuse = getUser(s_name);
        return ixuse.getPassword().equals(s_password);
    }
    public static IXpresUser getUser (String name) {
        // Some code that might return null
    }
}
// Login.java
public class Login {
    public static boolean isValidLogin (String name, String password) {
        boolean valid = true;
        try {
            setDatabaseConnection ();
            valid = XpresUserFinder.isUsernamePasswordCorrect (name, password);
        } catch (Throwable e) {
            e.printStackTrace();
        }
        return valid;
    }
}
```

If `UserFinder.getUser()` returns `null`, `UserFinder.isUserNamePasswordCorrect()` will throw a `NullPointerException`. This exception will be caught by `Login.isValidLogin()`, and the latter method will return `true`. This type of mistake is very basic, but also very easy to commit if you're not being careful. In this case, the valid `is_valid` should have been initialized to `false` to protect against the possibility of exceptions. This underscores that when working with exceptions, you need to be thinking about a more complex flow structure than you do for programs without exceptions.

The method `isUserNamePasswordCorrect` is also poorly behaved, and should include a check for `ixuse != null`. Clients of that method may only expect that method to return `true` or `false`, not to throw an unchecked exception.

To bolster security, you should always double-check that the places where the method is called can correctly handle any possible uncaught runtime exception.

Uncaught Runtime Exception Prevention

The only real way to prevent these security vulnerabilities is to identify all possible uncaught runtime exceptions early in the development lifecycle, then examine and modify the code to ensure that it does not provide any opportunities for hackers.

AEP's standard unit testing practice is designed to expose these uncaught runtime exceptions. Each AEP unit testing tool (such as Jtest, C++Test, and .TEST) is designed to automatically examine the code under test and determine how to test it. Its goal is to exercise each code unit with a wide variety of permissible inputs and then alert you to the potential runtime exceptions

that could occur with those inputs. If it identifies any inputs that could cause an uncaught runtime exception, it reports the inputs as well as the type of exception that the inputs produced.

To ensure that hackers do not exploit uncaught runtime exceptions, you must identify the potential uncaught runtime exceptions, then modify the code to ensure that they do not occur when the application is released/deployed. We recommend that you respond to all of the uncaught runtime exceptions that the AEP tool reports before you start working on other code. The appropriate response to take for each reported exception varies based on the nature of the exception.

For example, if an uncaught runtime exception is reported for a Java method that is behaving incorrectly (i.e., if that method should not throw an exception for the given arguments), you should modify the method's code so that it behaves correctly. The benefit of this repair is that the code will behave correctly and you will prevent potential instability, incorrect results, security problems, and/or crashes. This is the case described in the previous example.

If an uncaught runtime exception is reported for a Java method that will never receive the exception-causing arguments and/or is not intended to handle those arguments, you should either fix the code so that it throws an `IllegalArgumentException`, or add a Design by Contract comment that specifies the valid arguments. Either way, you prevent the exception and the associated security vulnerabilities.

If an uncaught runtime exception is reported for a Java method that is expected to throw an uncaught runtime exception, you should verify that the exception cannot be exploited by attackers, then document the method to clarify that the exception is intentional and that someone has verified that the exception will not cause a security vulnerability.

Another recommended preventative measure for crucial security methods, such as the `isUserNamePasswordCorrect()` method in the previous example, is to always add to the client code a `try/catch` to handle any possible exception.

Final Thoughts

With security, prevention is the key to success. You need to ensure that code is written so that later code modifications and reuse will not make the software vulnerable to attack. AEP—in particular, the practices we referenced in this paper—guides you through the process of preventing security vulnerabilities.

Preventative practices typically do not receive the respect they deserve, and almost every preventative practice is doubted and ignored by at least some people. For instance, ask a smoker his or her opinion about the value of preventing lung cancer and heart disease. Nevertheless, when preventative strategies are used in concert with detection-focused strategies, they allow you to dramatically reduce the risk of suffering from the problem you are trying to avoid, and can save you considerable time, effort, and resources by preventing something that is difficult to diagnose and remove.

Obtaining Additional Information

The security best practices discussed in this paper are a key part of Parasoft's Automated Error Prevention (AEP) methodology, a methodology that makes software work by automatically applying the lessons learned from your own mistakes and other IT groups' mistakes. An overview of AEP, as well as details on how your team can leverage AEP to prevent errors-- including security errors-- throughout the software development lifecycle, can be found on the Parasoft Web site at http://www.parasoft.com.

Contacting Parasoft

USA

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: +44 (020) 8263 2827
Germany: Tel: +49 7805 956 960
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com

Last Updated 2/19/04