An Integrated Multi-Agent Testing Tool for Security Checking of Agent-Based Web Applications

Fathy E.Eassa, M.Zaki, Ahmed M. Eassa and Tahani Aljehani
Software Engineering and Distributed Systems Research Group members
King Abdul-Aziz University
KSA
fathy55@yahoo.com azhar@eun.eg em_eassa@hotmail.com h-hanoo@hotmail.com

Abstract: - In this paper, an integrated multiagent testing tool, is presented. Such tool comprises static analyzer, dynamic tester and an integrator of the two components for detecting security vulnerabilities and errors in agent based web applications written in Java. The static analysis component analyzes the source code of the web application to identify the locations of security vulnerabilities and displays them to the programmer. Consequently, dynamic testing of the web application is carried out. Here, a temporal-based assertion language is introduced to help in detecting security violations (errors) in the underlying application. The proposed language has operators for detecting SQL injection and cross-site scripting, XSS, security errors.
The dynamic tester consists of two components: instrumentor (preprocessor) and run-time-agent. The instrumentor has many modules that have been implemented as software agents using Java language under the control of a multi agent framework. The agents of the instrumentor are: static analyzer agent, parser agent, and code converter agent. Moreover, an integrator for integrating both static and dynamic analyses is employed. Eventually the implementation details of IMATT are reported.

Key-Words: - web applications security testing, static testing, dynamic testing, temporal logic, assertion languages.

1 Introduction

In fact web applications represent a considerable share of software products. Such applications are continuously promoted using various software technologies. The promotion, as such, has led to web applications that are based on multiagent systems to provide: 1) user friendliness, 2) intelligent search and 3) better communications. Unfortunately, those web applications are subject to different attacks. This paper presents an integrated MultiAgent Testing Tool, IMATT, to facilitate static and dynamic testing procedures for finding out the security flaws, if any. In fact, the majority of the software testing tools are generic [2,23,25] in the sense that they are working independent of the style of the program under test. However, recently Centonze et al [2] have presented a tool named AEC for testing component based programs where the peculiarities of the program components are considered. Here we went a step further in this direction, where IMATT extends AEC and introduces, an agent based tool for testing large agent based Web applications (which are beyond component based programs) against security flaws. IMATT could be used with the following pragmatic advantages:

1. IMATT is homogeneous in the sense that both static and dynamic components are model based where the static analysis model is based on a set of grammar rules while the dynamic analysis model is based on temporal logic assertions in addition to a set of behavioral dynamic responses.
2. Integration of static and dynamic analysis, via path concatenation, enables the discovery of both intra and inter vulnerabilities.
3. Web applications allow intervention; consequently different scenarios for the same application can be generated. It is essential to check out the liveness of each scenario in order to guarantee the application ability to reach its goal. This is carried out by making use of temporal logic formalism.

Agent based web applications, Fig. 1, can be attacked (consequently protected at various levels). To be specific and to clarify the scope of IMATT, the MultiAgent, MAS, web application levels are pointed out as follows.

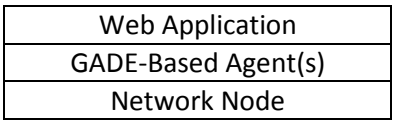

Fig. 1: Agent-based web application

1. Network node (site) level: where both attacks and protection mechanisms are network oriented and they are out of scope of this paper.
2. MAS environment (GADE) level: where malicious agent(s) could be introduced to attack the web application. At that level, the agent security is the responsibility of GADE-S that can allow authentication, authorization and integrity. Accordingly, IMATT is not involved.
3. Web application level: where IMATT is utilized to check out the underlying application.

Thus IMATT is a special purpose security testing tool that satisfies:

- The close fitting testing approach [1].
- Soundness (from static analysis), precision (from dynamic analysis) and flexibility by making use of a group of GADE agents for building up the instrumentor.
- IMATT can be easily involved in a continuous testing integration process [2, 3, 4, 5, 6], where iterating first one analysis, then the other is more powerful than performing either one in isolation [7].

There is a common agreement that attacks aimed at web applications represent most of today attacks [8], therefore the major types of such attacks are considered here. Namely, SQL injection [9, 10, 11] and cross site scripting, XSS [12, 13, 14], are adopted for their popularity, however, many other attacks could be illustrated in the same manner.

The rest of the paper is organized as follows. Section two is concerned with the related work while section three is concerned with the proposed architecture of IMATT. The implementation and testing of the tool are discussed in section four. Section five is concerned with the conclusion.

2 Related Work

Currently, there are several generic tools such as NuSVM, FDR2, ITS4, CHESS and NESSUS that could be exploited for program (code) analysis. Although they are widely used, such tools will not be considered here because they lack integration and their application domain is different. To be specific, IMATT will be only related to the class of tools that:

- Combines both static and dynamic code analyzes.
- Can be applied for Web application written in Java or an equivalent language.
- Can be devoted basically for detecting security vulnerabilities.
- Performs either model checking or any other sound approach to get decision.

The work of Centonze et al [2] has presented a proposal for combing static and dynamic analysis for automatic determination of database access control polices. Their tool could be applied on programs that are executed on stake-based access control systems such as Java. In their proposal the static analysis models the execution of the program taken into account native methods, reflection and multi-threading. In addition, the dynamic analysis can refine the potentially conservative results of the static analysis. The authors have implemented their analysis framework in a tool called Access Content Explorer, ACE. Such tool allows for automatic and precise identification of access-right requirements and library code location that should be made privilege-asserting to prevent any client code from requiring extra-access-rights.

An extension to the well-known tainted-mode model has been presented to afford inter-module vulnerabilities detection by Petukhov et al [8]. The authors have applied their proposal on web applications using dynamic analysis with penetration testing. Their automatic analyzer avoids the drawbacks of the manual-based code review recommended by OWASP (Open Web Application Security Project).The main contributions of that analyzer are:

- Improvement of classical tainted mode model so that inter-module data flows could be checked.
- Automatic penetration testing by leveraging it with information from dynamic testing output.

Livshits et al [15] have exploited a Program Query Language to build up a static analyzer for finding out security flaws in Java application. Moreover the authors have extended their work to include both static and dynamic techniques to check out the underlying queries. The static analyzer, given by livshits et al [15] finds the potential matches conservatively using a context-sensitive, flow-insensitive, inclusion-based pointer alias analysis. In addition their dynamic analyzer instruments the sources program to catch the

security violations when the program runs to perform user specified actions. By making use of these techniques, an analyzer has been designed and implemented to detect security flaws, resource leaks and violations of the predefined rules.

In their recent work Keromytis et al[6] have presented MINESTRONE as an architecture that integrates static analysis, dynamic confinement and code diversification techniques to enable the identification of vulnerabilities in a third party software. In its present from MINESTRONE in written in C/C++ and it seeks to:

- Enable the immediate deployment of new software, and,
- Enable the protection of legacy software.

The authors approach is to insert extensive security instrumentation, while leverage program analysis that is aided by runtime data. Diversification techniques are used as confinement mechanisms that may achieve software fault isolation.

The fundamental problem being addressed by MINESTRONE is finding vulnerabilities in the underlying software. Its key idea to realize this goal is to make use of the static analysis to allow reliable instrumentation, while runtime data provides a focus on portions of the code that are heavily exercised or otherwise considered security critical.

The tool Apollo has been discussed by Artzi et al in [16]. It aims at finding bugs in Web applications using dynamic testing and explicit state model checking. The proposed technique generates tests automatically, runs the tests capturing logical constraints on input and reduces the condition on the inputs to failing tests [16]. Thus Apollo provides test inputs for underlying application and validates that the output conforms to the predefined specification.

In all of the above mentioned tools no agents are considered or involved in either the Web application or the error-checker. In addition the integration process is always implicit.

3 Proposed Architecture of IMATT

This tool aims at finding both static and dynamic vulnerabilities in Web applications. Static vulnerabilities [9, 12] include SQL injection, cross-site scripting, XSS, while dynamic vulnerabilities are checked via the code coverage analysis using various metrics. The two approaches are similar in that they are model-based i.e. in both of them, vulnerability conditions are formally specified by the static tool. The dynamic tool takes the locations of the vulnerabilities and monitors if there are security violation during the web execution, Fig.2.

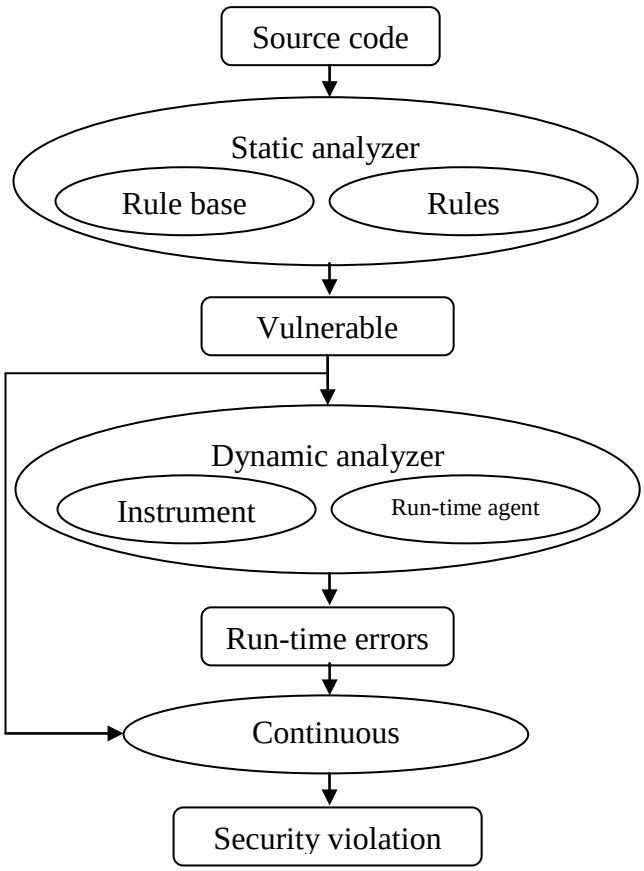

Fig. 2: IMATT architecture.

3.1. Static Vulnerabilities

Once malicious data has entered a Web application an attacker can use one of the following techniques (among others) to accomplish the expected breach.

3.1.1. SQL Injection

It is one of the well known security Vulnerabilities found in Web application. It is caused by unchecked user input being passed to aback-end database. The hacker may embed SQL commands into his data sent to the application.

Many SQL injections can be practically avoided with the use of better API's. Also, J2EE provides the prepare statement class, that allows specifying an SQL statements template capable for indicating statement parameters.

3.1.2. Cross-Site Scripting, XSS

It occurs when dynamically generated Web pages

display input that has not been properly validated [12]. An attacker may hide a malicious JavaScript code into such pages. When executed on the user machine, these scripts can breach the user account credentials. At the application level, echoing the application input back to the browser enables cross-site scripting.

3.2. Static Analysis

In its general form the static analysis problem should include object propagation problem [18, 19, 20, 21] with three types of description source descriptors, destination descriptors and derivation descriptors.

Source descriptors of the form <m,n,p> to specify ways in which user data can enter the program, where , m is a source method , n is parameter number and p is an access path to be applied to argument n to obtain the user-provided input. Destination descriptors have the same from with, m is a destination method, n is argument number and p is an access path to be applied to that argument.

Derivation descriptors have the form $<m,n_s,p_s,n_d,p_d>$ to specify how data probates between the program objects. In this case, m represents a derivation method; a source object is given by argument number n_s and access path p_s. A destination object is given by argument number n_d and access path p_d. Such descriptor specifies that at a call to method m, the object obtained by applying p_s to argument n_d is derived from the object obtained by applying ps to argument ns. Actually, in the absence of derived objects, to detect potential vulnerabilities, it is needed only to known if a source object is used at the destination.

In fact, derivation descriptors are used to handle the semantics of Java strings. Because Strings are immutable Java objects, string manipulation routines (concatenation in the underlying case) create new string objects, where contents are based on the original string objects. Actually, most Java programs use built-in string libraries and consequently share the same set of derivation descriptors [18].

The needed generalization may be achieved by making use of a simple syntax analyzer (parser) for data log queries to allow users to express vulnerability patterns in a friendly manner. Therefore, that approach will be relied upon in IMATT as it is explained in the following.

It should be noticed that the proposed approach does not replace the possibility of using the available Java security, API's and J2EE, instead it provides an affective extension for them to handle uncovered cases.

3.3. Dynamic Analysis

In order to detect the security violations during Web applications execution, an assertion language has been proposed. It is based on temporal logic to help in detecting security errors in a scope of the Web application. In addition, we have built a dynamic testing tool to instrument assert statements and detect security violations. In what follows the temporal assertion language is discussed.

3.3.1. Temporal Assertion Language

In order to detect the run time security vulnerabilities and error that occurs in Web applications, we introduce special language based on the temporal logic. We describe this language using Backus Naur Form (BNF). In this language we use the temporal logic operators (Always, Next, Eventually, Until). Also, the language has another two operators for detecting the security vulnerabilities (SQL, XSS).

As shown in the following Fig. 3, our assertion language has six temporal assert statements [Always, Eventually, Next, Until, XSS, SQL]. All of these assert statements (except next) are coupled with end-assert statements, thus enabling the tester to control the scope of the assert statement. Fig.3 shows the Java-based temporal assert statements.

```
Assert_Statemnt : Jweb_comment Jweb_Label "Assert"
Temporal_expression

Jweb_comment        : //

Jweb_Label          :[0-9][.0-9]+[a-zA-Z]

Temporal_expression:[ TL_operator ] ["("] (Logical
_expression )[")"]
                    : ["("] (Logical _expression)
[TL_operator] (Logical _expression)[")"]


TL_operator         : [ ] | ~| U | @ | SQL | XSS

Logical_expression  : (variable | constant )
Relational_connector (constant | variable )
                    : (Logical _expression)
Logical_connector (Logical _expression)

Relational_connector: >|<| ==| <=| >= |!=

Logical_connector   : && | || |!

End_assert_statment: Jweb_comment Jweb_Label
"Assert" "End"
```

Fig. 3: Java Temporal Assertion Language

The semantic of the temporal assertion language is determined according to choosing one of the temporal operators (Always, Next, Eventually, Until, SQL or XSS). Choosing those operators depends on the type of error that we want to detect. Suppose it is required to ensure that some variables never equal zero along the scope of certain code, then we use always operator, but if we want to check whether the input field contains SQL injection or not so we will use SQL operator. Such operators semantics are pointed out in the following.

1) Always (safety) properties: A temporal expression of this form // 1.1.A Assert [ ] (W) , specifies that W is always true, during the scope of the always assert statement. Note that the assert statement starts with double slash followed by label followed by Assert keyword and finally the condition (W).
2) Eventually (liveness) properties: The eventually operator (~) of this form // 1.1.A Assert ~ (W) is used to test that a specific condition (W) is satisfied at least once during the scope of the eventually assert statement.
3) Precedence properties: The until (U) temporal operators of this form // 1.1.A Assert T1 U T2. Can be used to assert that Task T1 will start when Task (T2) finishes. We can use this property to check race condition.
4) SQL properties: The SQL temporal operator of this form // 1.2.A Assert SQL (variables). We use this property to insure that the variables in the form are not injected with SQL attack.
5) XSS properties: The XSS temporal operator of this form // 1.2.A Assert XSS (variables). We use this property to insure that the variables in the form are not injected with XSS attack.

3.3.2. The Architecture of the dynamic testing tool

This section introduces the architecture of the dynamic tool. The programmer adds temporal assert statements to the source code of the agent-based web application in the position that he expects errors. The agent based instrumntor consists of set of agents. Agents detect the assert statements in the web application under testing and convert each one to the corresponding Java statements. The basic components in our dynamic testing tool are presented in Fig. 4.

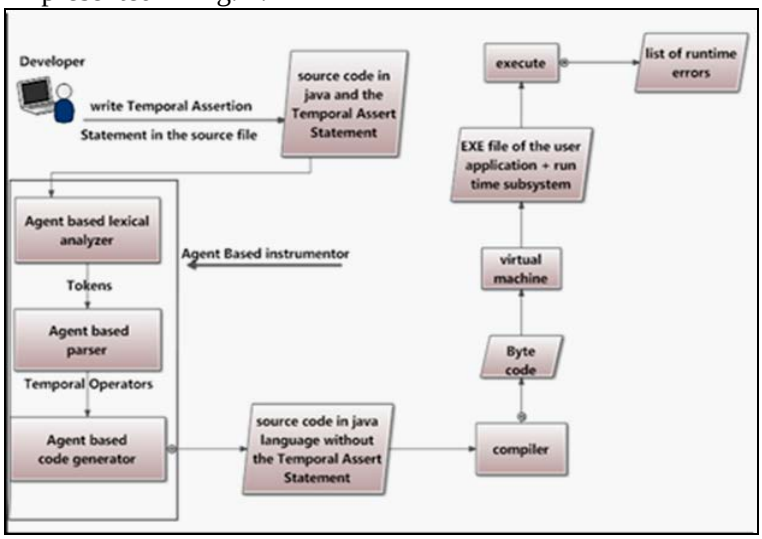

Fig. 4: Agent Based Dynamic Testing Tool Architecture

3.3.2.1 Agent Based Lexical Analyzer: The agent-based lexical analyzer reads the (java source file which has the temporal assert statements within the source code). Then this agent tokenizes the file to set of tokens which will be sent to the agent-based parser. The pseudo code of the lexical analyzer agent is shown in Fig. 5.

```
Show_Gui()
Choose_Folder()
foreach SourceFile in Folder
   Create DestinationFile
            WHILE (Line =SourceFile.ReadLine() !=null)
               IF Line has Assert
                        IF Line has Temporal Operator
                                 Send SourceFile to Parser
                                 Send DestinationFile To Parser
                     Send Line To Parser
                                 END IF
            Block ( )
                        END IF
         ELSE
                        Write Line in DestinationFile
 END WHILE
         Receive GeneratedcodeFile
         Copy GeneratedcodeFile To SourceFile
END Foreach
```

Fig. 5: The pseudo code of the lexical analyzer agent

3.3.2.2 **Agent Based Parser:** The parser reads the tokens and then decides whether the tokens are Java statements or assert statements. If they are Java statements, it will write it to the destination file which contains only the Java source code without the temporal assertion, otherwise if the statements start with double slash followed by the assert keywords and one of the temporal logic operators, then source code will be generated based on the kind of the temporal operators. The pseudo code of the parser agent is shown in Fig. 6.

```
Receive SourceFile
Receive DestinationFile
Receive Line
StrLine=Line
Declare Label
Declare Temporal
Declare Condition
Declare Agent
Array=StrLine.toCharArray()
FOR char in Array
 IF char=='('
   WHILE  char!=')'
              move char to condition
  END WHILE
  END IF
 END FOR
Array = StrLine.split ([ " "]+)
Label=Array[1]
Temporal =Array[2]
IF Temporal==[]
 Agent=Alwyas_CodeGeneration
ELSE IF Temporal=U
 Agent=Until_CodeGeneration
ELSE IF Temporal=@
 Agent=Next_CodeGeneration
ELSE IF Temporal= ~
 Agent=Eventually_CodeGeneration
ELSE IF Temporal=SQL
 Agent=SQL_CodeGeneration
 ELSE
 Agent=XSS_CodeGeneration
Send SourceFile To Agent
Send DestinationFile To Agent
Send Label To Agent
Send Condition to Agent
```

Fig. 6: The pseudo code of the parser agent.

3.3.2.3. **Agent Based Code Generator:** Depending on the temporal logic operators, this agent will generate the code for each temporal assert statement. The pseudo code of the code generation agent is shown in Fig. 7.

```
Receive SourceFile
 Receive DestinationFile
 Receive Label
 Receive Condition
 WHILE ( Line = SourceFile.readLine()!=null)
  IF Line==Assert
    Continue
   IF Line ==Label
    Break
  Array1=GetVaribles(Condition)
  Array2= Tokonize(Line)
  FOR i=0 to Array1.lenghth
   FOR j=0 to Array2.lenghth
    IF Array1[i]==Array2[j]
     Write Line to DestinationFile
      Write the corresponding java code To DestinationFile
     END IF
     ELSE
      Write Line to DestinationFile
  END FOR
    END FOR
 END WHILE
Send GeneratedJavaCode to MainAgent
```

Fig. 7: Code generation agent pseudo code.

3.4. Integration of Static and Dynamic Analyzers

Given a large program, it may be impractical to identify, manually, security failures. However, by integrating static and dynamic analyses [25], IMATT can soundly model the program behavior to identify the security vulnerabilities. Consequently, using the dynamic analysis would handle second order (indirect) run-time attacks.

While theoretically sound, in practice the static analysis may be unsound for the following reasons:

1) Multi-language code: A Java program may trigger the execution of methods written in C and executed directly on the operating system. A static analyzer for Java will not be able to model C functions. As a result the analysis will fail.
2) Reflection: which is a mechanism that enables code to dynamically manipulates fields and methods of loaded classes. Modeling reflection through static analysis is unsound since the type of object obtained through reflection is only available at runtime.

In fact neither static nor dynamic analysis can independently guarantee the identification of all security vulnerabilities. Actually, dynamic analysis suffers from the fact that:

- It needs a set of functional or security rules that may be practically unavailable [22].
- It needs a set of attacks like those used in the real world. In addition it needs a collection of temporal information.
- It is destructive since it may perform attack execution

IMATT integration, Fig. 8,consists of two analyzing modules: static and dynamic, where each analyzer is designed as a multi-agent subsystem. The static analyzer agents read the Java-based web application, and analyze it to identify a list of security vulnerabilities. Based on the list of identified vulnerabilities,the user (programmer) inserts some assert statements in the web application and creates new web application file that contains java statements and assert statements. The dynamic testing agent reads the new file and instrument it, so that it can cover all security violation at various levels. Eventually it displays the violations,if any of them is reveald during Web application execution.

In IMATT, the need to integrating static and dynamic analyses is a must. This is because the fact that agents, specially mobile ones use extensively ‘reflection’ in their programing pardigm. Actually, modelling reflection by making use of static analysis is unsound since the type of underlying objects that are obtained through the reflection is identified only at run time.However, the dynamic analyzer uses reflection to load classes , create objects and invoke the required methods.

Accordingly, the process of creating a testcase is automated (but not eliminated).

On the other hand , relying on pure dynamic analysis is not sufficient because of its dependency on the test cases. In practice it is usual that some execution paths, along with the previledged rights to execute those paths may remain undiscovered until the code deployment phase. This yields an incomplete cover for the program under test, consequently unsoundness is arised due to the absence of a formal cover that should be generated by the selected test cases.

IMATT integrator, Fig.8, has several essential features that can be pointed out in the following:

- It tackles the reflection problem(s) by conservatively locating the suspected agent using the static analyzer, and then the dynamic analyzer is employed to refine the obtained conservative results, i.e. to extract the runtime rule(s) violation.
- A Java temporal assertion language is implemented with well defined semantics. Such language combines on a formal basis, temporal logic and application oriented operators.
- One of the roles of the proposed integrator is to eliminate false alarms, i.e. when the static analyzer might report a false alarm (due to security sensitive action) the dynamic analyzer that utilizes the coverage of the underlying program methods can eliminate the statically detected false alarms.

For IMATT each solution is executed in three steps.

1) The static analyzer discovers the call that may cause security vulnerability and determines its location (agent)
2) At run-time the dynamic analyzer checks out the vulnerability locations of the underlying agent to discover the method that can yield a breach. In addition it logs the underlying operation in a special file that might be parsed for security holes.
3) From steps 1 and 2 the integrator, Fig.8, exploits continuous integration agent which is coupled with both static and dynamic analyzers in order to find out the corrupted class which is responsible for the security violation problem.

Also, the security side effects can be discovered and detected. For convenience such details are moved to Sec.4 , where illustration of IMATT implementation, using several experimenal examples, is given.

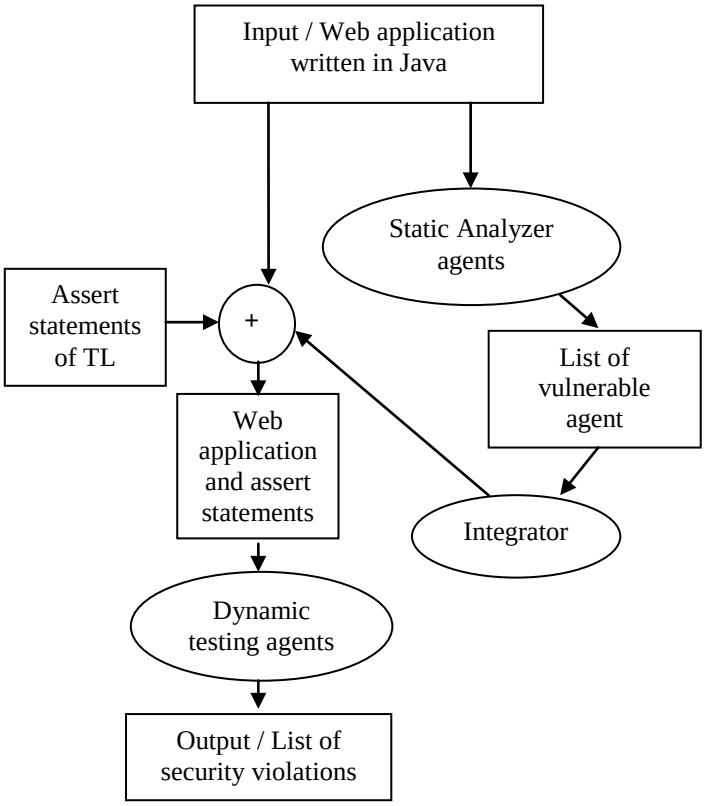

Fig. 8: The Integration of static and dynamic tools.

4 Tool Implementation and Testing

All agents of the testing tools are written in Java programming language. In addition JADE [24] as a middleware that facilitates the development of multi-agent systems is used to manage and run the agents of IMATT

4.1 Code Generation for SQL Injection and XSS

SQL Code Generation Agent: When the agent receives the source file , destination file , and the pointer to both files with the condition and label , it starts to extract the variables from the conditions and then starts reading the source file from the pointer until it finds the label followed by word "END". When the agent reads the source file each line has any one of those variables, the agent will insert run time method called hasSQL() in the destination file after the java statement which has one of those variables the method which will take variables as the arguments analyzes the variables to ensure no SQL injections , otherwise the agent will

write the java statement in the destination file .After reaching the end of the assert statement, the control flow will return back to the lexical analyzer which will continue reading the source file from where the code generation ended reading and the procedure will be repeated again when the lexical analyzer agent catches any temporal assert statements . We use the SQL temporal operator when we want to detect SQL attack.

A similar XSS code generation agent can be obtained by replacing SQL by XSS.

4.2. Testing of Web Applications

For testing Web applications, the Web application under testing is inserted by temporal assert statements. After that the instrumentor part of IMATT instruments the Web application, where translates each temporal assert statement based on the semantic of the temporal operator to Java statments. The instrumented Web application is compiled and executed for detecting any security attack. To clarify the nature of IMATT more examples that are concerened with the implementation details are given in what follows.

Example 1: Detection of SQL Injection using the SQL Operator:

•*The problem:*

Suppose we have Web application of a company, where there is a service that allows us to retrieve information of an employee from the database by giving his first name . Suppose "John" is entered and "submit" button is pressed, information of the employee "John" is retrieved and displayed as shown in Fig. 9. Asumme an attacker would like to get information of all employees in the company, he will insert John ' OR '1'='1 in the field of employee's name, so the query will be *select * from employees where firstname='" + John ' OR '1'='1 + "'";* due to this SQL injection and because the 'OR' expression is always true, information of all employees are retrieved and displayed as shown in Fig. 10. This allows an attacker to take information of all employees. Using the same technique attackers can inject other SQL commands which could extract, modify or delete data within the database.

- *Solution of the problem*

In order to detect the SQL injection , a temporal assert statement is inserted in the agent-based Web application to check the fields of the form. In the code of Fig. 11, the inserted temporal assert statement is // 1.2.A Assert SQL (user), where the (user) in this statement will be the data entered by the client or attacker.

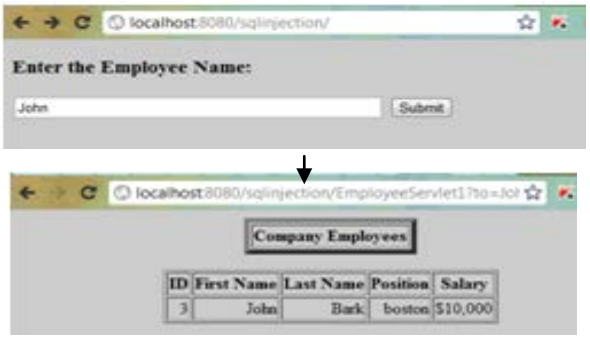

Fig. 9: The record of John

Fig. 10: Information of all employees due to SQL Injection

The code of Fig. 11 is instrumented by agents of the dynamic analyzer to generate a pure Java code as shown in Fig. 12. The generated Java code contains a method called hasSQL() that takes the fields of the form as an argument and checks if the field has SQL attack characters or not.

```
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        // 1.2.A  Assert SQL ( user )
        user=request.getParameter("to").toString();
        // 1.2.A END

        String driver ="org.apache.derby.jdbc.ClientDriver";
String docType =
"<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
"Transitional//EN\"\n";
String title = "Company Employees";
out.print(docType +
"<HTML>\n" +
"<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
"<LINK REL='STYLESHEET' HREF='./css/styles.css'\n" +
" TYPE='text/css'>" +
"<BODY bgcolor=#CCCCCC  ><CENTER>\n" +
"<TABLE CLASS='TITLE' BORDER='5'>" +
" <TR><TH>" + title + "</TABLE><P>");
showTable(out);
out.println("</CENTER></BODY></HTML>");
        // processRequest(request, response);
    }
```

Fig. 11: Shows SQL injection and inserted assert statement in Web Application

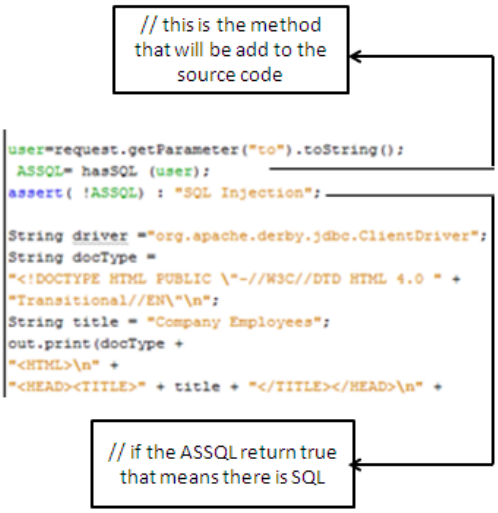

Fig. 12: Output of Temporal Assert Statements Instrumentation.

• *Executing the Web Application after instrumentation:*

After executing the program in Fig. 12, and entering (John ' OR '1'='1) in the field of employee, we see in Fig. 13, the assertion exception arises after the detection of SQL injections.

Fig. 13: SQL Injection violation that is detected by the dynamic analyzer

Example 2: Detecting XSS Attack by using XSS operator:

• *The problem:*

Suppose Myspace Web site of a Web application has been singed up by a malicious user and in his profile page the following script has been added. So, every time a visitor visits the profile the script is gotten and annoyed.

```
<script>alert('Hello World');</script>
```

Now suppose that the problem get bigger where a code has been added in the comments of the site as shown in the following statement.

```
<a href="/usercp.php?action=logout">A webpage about cats</a>
```

So, every time the users click on this link they will visit web site about cats, but they will be logged out of the web site and that's so annoying.

The problem will be worst if the attacker has injected script which steals user cookies. So, every one visit the guess book, he will be redirected to a page at attacker's site. The cookies from MySpace's browser session have been transmitted to attacker's web server as part of the URL. This will allow the attacker to steal the pass word and the username of the administrator of the web site, and the attacker gives himself administrator access, or start deleting content.

And now come to the most dangerous problem if the attacker could have used a JavaScript link to trick users into sending sensitive information to his server

```
<a href= " javascript:location.replace ('http://rickspage.com/?secret='+document.cookie) "> A Webpage about dogs</a>
```

If users clicked that link, as they probably do often, their session ID would be transmitted to attacker's server. Fig. 14 and explains the problem.

Fig. 14: The script to steal user session has been added

• *Solution of the problem:*

In order to detect the XSS attack, a temporal assert statement **// 1.2.R Assert SQL (name, email, comm)** has been inserted to check the fields of that form as shown in Fig. 15; the name , email and comm are the form fields.

The code in Fig. 15 is instrumented by the agents of the dynamic analyzer to generate a pure Java code that contains a method called hasXSS() as shown in Fig. 16. The data of the fields of the form are received and checked by the hasXSS() method during the Web application execution.

• *Testing the Web Application after instrumentation:*

The code in Fig. 16 has been compiled and executed. The input that contains XSS attack has been entered. The XSS attack has been detected by the tool Fig. 17.

In order to emphasize the relative merits of IMATT, its performance upon compacting versus should be compared practically with similar

analyzers.

However, such task could not be accomplished due to Lack of published quantitative information of the performance of such similar products

```
String title = "Reading All Request Parameters";
// 1.2.R Assert XSS (name,email,comm)
String name=  request.getParameter("lastName");
String email=  request.getParameter("email");
String comm=  request.getParameter("commen");
  Enumeration paramNames = request.getParameterNames();
  while(paramNames.hasMoreElements()) {
    String paramName = (String)paramNames.nextElement();
    out.println("<TR><TD>" + paramName + "\n<TD>");
    String[] paramValues = request.getParameterValues(paramName);
    if (paramValues.length == 1) {
      String paramValue = paramValues[0];
      if (paramValue.length() == 0)
        out.print("<I>No Value</I>");
      else
        out.print(paramValue);
    } else {
      out.println("<UL>");
      for(int i=0; i<paramValues.length; i++) {
        out.println("<LI>" + paramValues[i]);
      }
      out.println("</UL>");
    }
```

Fig. 15: Inserting temporal assert statement in the Web application.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();

String title = "Reading All Request Parameters";

String name=  request.getParameter("lastName");
ASSXSS= hasXSS(name);
assert(!ASSXSS):"XSS Attack";
String email=  request.getParameter("email");
ASSXSS= hasXSS(email);
assert(!ASSXSS):"XSS Attack";
String comm=  request.getParameter("commen");
ASSXSS= hasXSS(comm);
assert(!ASSXSS):"XSS Attack";

  Enumeration paramNames = request.getParameterNames();

  while(paramNames.hasMoreElements()) {

    String paramName = (String)paramNames.nextElement();

    out.println("<TR><TD>" + paramName + "\n<TD>");
    String[] paramValues = request.getParameterValues(paramName);
    if (paramValues.length == 1) {
      String paramValue = paramValues[0];
```

Fig. 16: The generated code after the instrumentation.

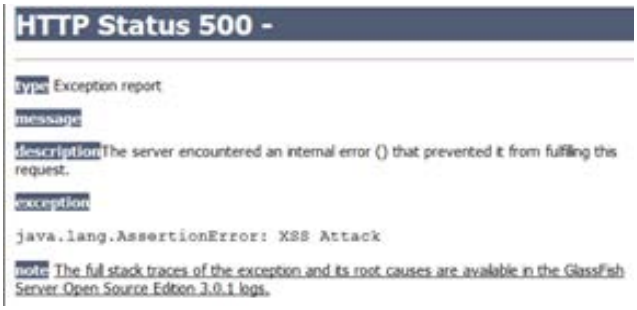

Fig 17: Assertion Exception after detecting the XSS attack

5 Conclusion

This paper presents IMATT as a special purpose integrated multiagent tester that integrates both static and dynamic testing components to check out the security of agent based Web applications. IMATT has been built up using software agents.

The static component consists of a rule-base and a code checker while the dynamic component consists of instrumentor and a run-time analyzer. In order that such analyzer can handle different scenarios of the Web application it makes use of temporal logic to examine the application under test. The integrator integrates the results of both components to get a decision for either intra or inter attacks. In the present state, the temporal assert statements are inserted manually in the Web application, however, in future, it is planned to assign an intelligent agent that can be able to insert such statements automatically.

References:

[1] Baca D, Peterson K, Carlsson B and Lundberg L, Static Code Analysis to Detect Software Security Vulnerabilities-Does Experience Matter?, Availability, Reliability and Security International Conference, Blekinge, March 2009.

[2] Centonze P, Flynn R. Pistoia M, Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies, Proceedings of the 23rd Annul Computer Security Applications Conference, 2007.

[3] Tzermias Z, Sykiotakis G, Polychronakis M and Markatos E, Combining Static and Dynamic Analysis for the Detection of Malicious Documents, , available at

[4] http://dcs.ics.forth.gr/activites/papers/mdscan.eurosec

[5] Lam M S, Martin M, Livshits VB and Whaley J. Securing Web Applications with Static and Dynamic Information Flaw Tracking, Available at http://suif.stanford.edu/papers/pepm08.pdf.

[6] Blazarot D, Marco C, Felmetsger V, Javanovic N, Kird E, Kruegel C and Vigna G, Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications, SP '08 Proceedings of the 2008 IEEE Symposium on Security and Privacy, 2008,pp 387-401.

[7] Keromytis A, Stolfo S, Yang J, Stavrou A, Ghosh A, Angler D, Dacier M, Elder M and Kienzle D, The MINESTRONE Architecture: Combining Static and Dynamic Analysis

Techniques for Software Security, Available at http://www.cs.columbia.edu/~angelos/Papers/2011/minestrone-syssec.pdf

[8] Johnson M, Ho C-W, Maximilen M and Willlams L, Incorporating Performance Testing in Test Driven Development ,IEEE Software, May/June 2007,pp 67-73.

[9] Petukhov A and Kozlov D, Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing, Application Security Conference, Ghent-Belgium, May 2008, pp 1-16.

[10] Khochare N, Chalurkar S, Kakade S and Meshramm B, Servey on SQL Injection Attacks and their Countermeasures, International Journal of Computational Engineering and Management, October 2011, pp 111-114.

[11] Muthuprasanna M, Wiek and Kothari S, Eliminating SQL Injection Attacks - A Transparent Defense Mechanism, available at http://home.engineering.iastate.edu/~muthu/papers/cnf08.pdf

[12] Balasundaram I and Ramaraj E, An Approval to Detect and Prevent sql Injection Attacks in Database Using Web Service, International Journal of Computer and Network Security, Vol. 11,No. 1, January 2011,pp 197-205.

[13] Bisht P and VenkatataKrishnan, XSS-GUARD: Precise Dynamic Prevention of Cross Site Scripting Attacks, Available at http://www.cs.uic.edu/~Pbisht/XSSGuard_DIMVA2008_bisht-pdf.

[14] Di_Lucca G. A., Fasalino A. R., Mastoinni M. and Tramontana P., Identifying Cross Site Scripting Vulnerabilities in Web Applications, Proceedings of 6th IEEE International Workshop on Web Site Evolution, September 2004, pp 71-80.

[15] Shanmugam J, Ponnavaikko M, Cross Site Scripting: Latest Developments and Solution, International Journal of Open Problems in Computer and Mathematics, Vol.1, No.2, Sepetmper 2008, pp 101-121.

[16] Livshits VB and Lam MS, Finding Security Vulnerabilities in Java Applications with Static Analysis, Available at http://Usenix.Com/events/sec05/tech/full_papers/Livshits.

[17] Artzi S,Kiezun A,Dolby J, Tip F, Dig D, Paradkar A and Ernst M, Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit State Model Checking, IEEE Transaction on Software Engineering, Vol. 36, No. 4, July/Aug 2010,pp 474-494.

[18] Huang Y, Yu F, Hang C, Tsai C, Lee D and Kuo S, Securing Web Application Code by Static Analysis and Runtime Protection, Proceedings of the 13th International Conference on World Wide Web, ACM, New York, 2004 pp 40-52.

[19] Jovanovic N, Kruegel C, Kirda E, Static analysis for detecting taint-style vulnerabilities in web applications, Journal of Computer Security, Vol 18,2010,pp 861-907.

[20] Guizani W, Marion J and Reynaud-Plantey D, Server-Side Dynamic Code Analysis, 4th International Conference on Malicious and Unwanted Software (MALWARE), October 2009, pp 55 - 62.

[21] M 86th Security Paper, Real-time Code Analysis: Proactive Protection Against New and Dynamic Malware Threats, available at http://www.m86security.com/documents/pdfs/white_paper

[22] Fu X, Lu X, Peltsverger B, Chen S, Qian K, Tao L,A static analysis framework for detecting injection vulnerabilities, Proceedings of the 31st Annual International Computer Software and Applications Beijing, July 2007, pp 87–96.

[23] Bessey A, Block K, Chelf B, Chou A, Fulton B, Hallem S, Henri-Gros C, Kamsky A, Mcpeak S and Egler D, A Few Billion Lines of Code Later : Using Static Analysis to Find Bugs in The Real World, CACM,Vol.53,No.2,2010,pp 66-75.

[24] Halfond, W and Orso A, WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation, IEEE Transaction on Software Engineering, Vol. 34, No. 1, January/ February 2010,pp 65 - 81.

[25] Giovanni C, JADE Tutorial , Available at

[26] http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf

[27] Ernst M, Static and Dynamic Analysis: Synergy and Duality, Available at http://www.cs.nmsu.edu/~jcook/woda2003/papers/Ernst.pdf.