jModex: Model Extraction for Verifying Security Properties of Web Applications

Petru Florin Mihancea   Marius Minea
LOOSE Research Group
Politehnica University of Timișoara   and   Institute e-Austria Timișoara, Romania
Email: {petru.mihancea, marius}@cs.upt.ro

***Abstract*—Detecting security vulnerabilities in web applications is an important task before taking them on-line. We present JMODEX, a tool that analyzes the code of web applications to extract behavioral models. The security properties of these models can then be verified with a model checker. An initial evaluation, in which a confirmed security flaw is identified using a model extracted by JMODEX, shows the tool potential.**

I. INTRODUCTION

Due to the proliferation of services provided by companies to their customers via the Web, the software industry is confronted with increasingly complex problems while developing and evolving web applications. Since these programs manipulate usually sensitive information, security issues are definitely some of the most important ones.

To achieve its security requirements, an application uses dedicated mechanisms such as password authentication, cryptographic techniques, etc. Unfortunately, due to the increasing complexity of web applications, they also become more difficult to comprehend by developers. Thus, some *security vulnerabilities* may pass unobserved, enabling an intruder to bypass the security mechanisms used by the application.

To support identification of these vulnerabilities in code, the research community has developed different approaches. Some of them detect low-level vulnerabilities such as code injection. Others detect specific classes of missing checks [1], or try to infer logical invariants that are violated [2].

We focus on finding logical vulnerabilities using model checkers, given explicit specifications of the desired system properties. To maintain the advantage of verifying an actual implementation, our models are extracted from source code. We have developed the prototype tool JMODEX, which analyzes the code of a web application written using the JSP/Servlet technology and builds a behavioral automaton model for the system. This model is then expressed in the ASLAN++ modelling language, which enables its verification with security model checkers like CL-ATSE [3] or SATMC [4]. The approach is complementary to using such models for automated test generation [5].

II. MODELING IN ASLAN++

To illustrate the modelling of a system in ASLAN++, Listing 1 presents the central part of a model for a very simple web program. A client can send to the application a message (line 9) with two parameters: *RQP_action* and *RQP_user*. Possible actions are *login*, which can be performed as *admin* user (line 11) or as *guest* user (line 16), and *logout* (line 20). The application tracks the number of successful *login* actions using a global counter (lines 13 and 17). The counter can be *reset* by another action (line 25), but only if the client is logged in as *admin* at that moment. This security requirement is specified in the model using an assertion (line 26): when the *reset* action is performed, the *asAdmin* predicate must be *true*. This predicate is updated at lines 14, 18 and 22. We emphasize that these latter statements do not model the actual program but its expected behavior, the specification of the modeled system.

Listing 1. ASLAN++ Specification Example

```
1   entity Application(Actor:agent,RU:agent) {
2    symbols
3     RQP_action,RQP_user,SA_isAdmin:message;
4     asAdmin(agent):fact;
5    body {
6     SA_isAdmin := oNull;
7     while(true) {
8      select {
9       on (?RU*->*Actor:rExample(?RQP_action,?RQP_user)): {
10        select {
11         on (sadmin = RQP_user & slogin = RQP_action): {
12          SA_isAdmin := syes;
13          RExample_counter := rintAdd(RExample_counter,i1);
14          asAdmin(RU);
15         }
16         on (sguest = RQP_user & slogin = RQP_action): {
17          RExample_counter := rintAdd(RExample_counter,i1);
18          retract asAdmin(RU);
19         }
20         on (slogout = RQP_action): {
21          SA_isAdmin := oNull;
22          retract asAdmin(RU);
23         }
24         on (syes = SA_isAdmin & sreset = RQP_action): {
25          RExample_counter := i0;
26          assert onlyAsAdmin:asAdmin(RU);
27         }
28  }}}}}}
```

When verifying this model, the CL-ATSE model checker automatically produces the sequence of events in Listing 2 as a counter-example for our security requirement. In short, an intruder (named *i*) logs in as *admin*, makes another login as *guest* in the same session, and finally resets the counter.

Listing 2. The Attack Identified by CL-ATSE

```
<i>  *->*   Actor(1)     : rExample(slogin,sadmin)
<i>  *->*   Actor(1)     : rExample(slogin,sguest)
<i>  *->*   Actor(1)     : rExample(sreset,RQP_user(54))
```

Following this interaction scenario, the *reset* can be performed even by a client currently logged in as *guest*, highlighting a logic security vulnerability. The reason is that logins are

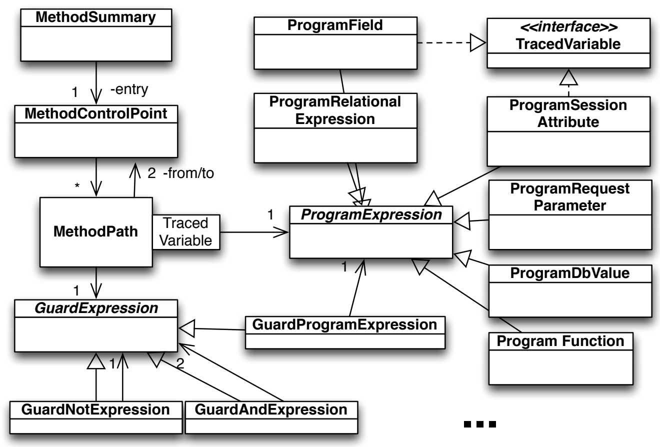

Fig. 1. A Fragment of the iSUMMARIZE Meta-Model

not tracked per session: variable *SA_isAdmin* has the symbolic value *syes* set during the first login of the attack scenario (line 12). But an intruder can ask for a *guest* login immediately after an *admin* login. Thus, *SA_isAdmin* should be reset to *oNull* not only on *admin* logout, but also when a *guest* logs in.

III. Model Extraction with JModex

To verify some security requirements of web applications using a checker like CL-ATSE, we first need a model of that program. We have built JMODEX, an ECLIPSE plug-in that automatically extracts an ASLAN++ model from the implementation of a JSP/Servlet-based web application. In the following we briefly describe how JMODEX achieves its goal.

A. iSUMMARIZE – *Extraction of Behavioral Automata*

The first task performed by JMODEX is to capture the behavior of each component (i.e., servlet) of the application in the form of an *extended finite state machine*.

A node in this automaton corresponds to a control point (e.g., the entry / exit point of a component or a loop header) while an edge corresponds to an execution path within the component. For each edge, the analysis determines a *guard* representing the condition which triggers the execution of the associated execution path, and a set of *updates* that capture the assignments to the relevant state variables on the same path.

Figure 1 presents a part of the meta-model used to represent these automata. It captures the various kinds of expressions in the Java language (e.g., *ProgramRelationalExpression*). However, we do not analyze the libraries used by an application: special entities model the semantics of the servlet API (e.g., assigning/using the value of a *ProgramSessionAttribute* or *ProgramRequestParameter*); SQL queries (e.g., returning a *ProgramDbValue* from the database); or library functions abstracted away as uninterpreted (*ProgramFunction*).

To build the automaton for a given component, JMODEX uses the WALA[1] library to extract the component call-graph and the control-flow graphs of each method. Next, JMODEX does a depth-first traversal of the call-graph and computes the automaton of each method such that a caller is processed after all its called methods. This is required because our analysis is inter-procedural and we in-line the automaton of a called method into the automaton of the caller (recursion is currently not supported). Consequently, when the call-graph traversal is finished, we obtain a single automaton for the entry method of the component and implicitly, of the entire component.

To build the automaton of a method, JMODEX traverses the control-flow graph depth first, processing each instruction backwards from the exit to the method entry. This determines all execution paths through the method together with their guards and updates. JMODEX uses large-block encoding [6] to collapse successive steps in the control flow graph and produce a model with fewer transitions. Figure 2 shows various phases of building the automaton, based on the code in Listing 3.

Listing 3. A Code Example

```
 1  public class Example ... {
 2   private static int counter =  0;
 3   public void _jspService(...)  ... {
 4    String action = request.getParameter("action");
 5    if ("login".equals(action)) {
 6     String user = request.getParameter("user");
 7     if ("admin".equals(user))
 8      request.getSession().setAttribute("isAdmin", "yes");
 9     if ("admin".equals(user) || "guest".equals(user))
10      counter = counter + 1;
11    } else if ("reset".equals(action)) {
12     String isAd = (String) request
13      .getSession().getAttribute("isAdmin");
14     if ("yes".equals(isAd))
15      counter = 0;
16    } else if ("logout".equals(action))
17     request.getSession().removeAttribute("isAdmin");
18    else ;  // NOP - added for better understanding
19   }
20  }
```

The analysis starts at the end of the method (line 19) and an initial automaton is built (see Figure 2A). It has a single edge with the *true* guard and no update. It is easy to see that there are 4 ways in which line 19 can be reached: from the end of the branches at lines 5, 11, 16 and 18. Consequently, the initial edge will be split such that one copy propagates backward through each of these distinct points (see Figure 2B).

Next, edge 1 is propagated backward through the false branch of the *if* in line 16. Since there are no operations, nothing happens. Similarly, edge 2 is propagated backward through the true branch of the same condition. This time JMODEX determines that a relevant state variable (i.e., session attribute) has been changed (line 17) and the corresponding update is added to edge 2. After the propagation of both edges, JMODEX observes that the condition in line 16 is responsible for triggering the execution of edge 1 or 2. Consequently, their guards are changed accordingly (see Figure 2C).

Next, edge 3 is propagated backward through the true branch of the *if* in line 11. First, it will be split in two because inside this branch there are two possible sub-paths: on one JMODEX determines an update to a relevant state variable (counter reset) while on the other there are no updates. The execution of these sub-paths is controlled by the condition in line 14 and thus their guards are modified accordingly. Both

[1]http://wala.sourceforge.net

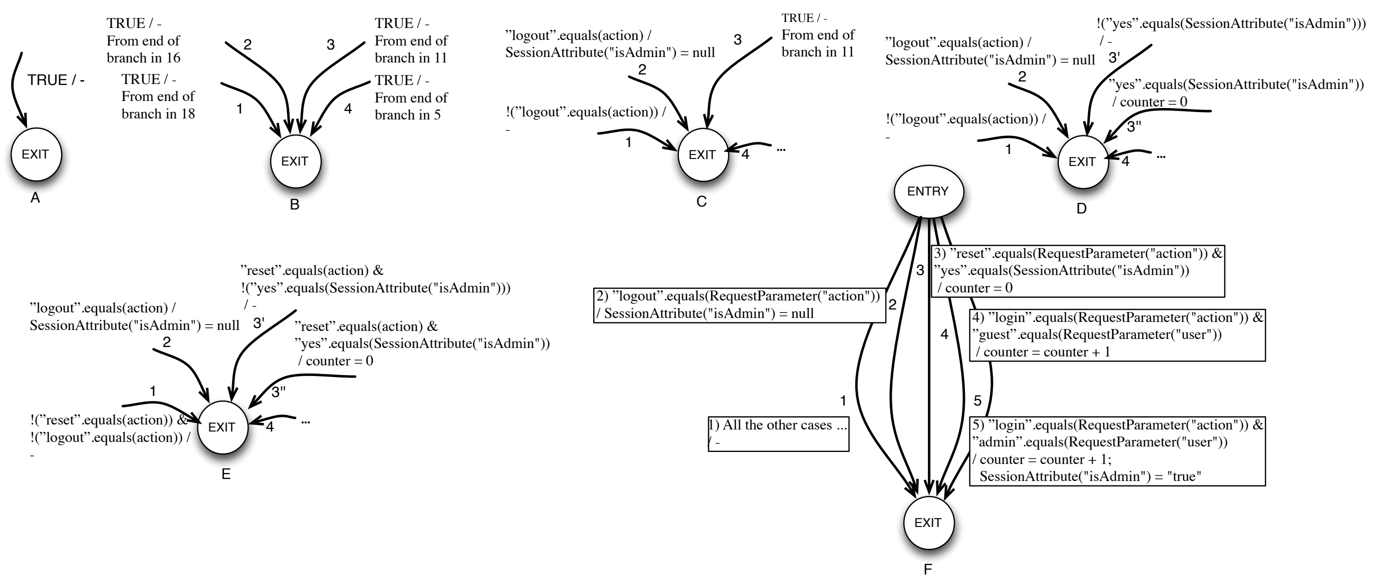
TRUE / -
EXIT
A
TRUE / -
From end of
branch in 16
2
3
TRUE / -
From end of
branch in 11
TRUE / -
From end of
branch in 18
1
4
TRUE / -
From end of
branch in 5
EXIT
B
"logout".equals(action) /
SessionAttribute("isAdmin") = null
2
3
TRUE / -
From end of
branch in 11
!("logout".equals(action)) /
-
1
EXIT
4
...
C
"logout".equals(action) /
SessionAttribute("isAdmin") = null
2
!("yes".equals(SessionAttribute("isAdmin")))
/ -
3'
"yes".equals(SessionAttribute("isAdmin"))
/ counter = 0
3"
!("logout".equals(action)) /
-
1
EXIT
4
...
D
"reset".equals(action) &
!("yes".equals(SessionAttribute("isAdmin")))
/ -
"logout".equals(action) /
SessionAttribute("isAdmin") = null
3'
2
"reset".equals(action) &
"yes".equals(SessionAttribute("isAdmin"))
/ counter = 0
1
3"
!("reset".equals(action)) &
!("logout".equals(action)) /
-
EXIT
4
...
E
ENTRY
3) "reset".equals(RequestParameter("action")) &
"yes".equals(SessionAttribute("isAdmin"))
/ counter = 0
2) "logout".equals(RequestParameter("action"))
/ SessionAttribute("isAdmin") = null
3
2
4
4) "login".equals(RequestParameter("action")) &
"guest".equals(RequestParameter("user"))
/ counter = counter + 1
1
5
1) All the other cases ...
/ -
5) "login".equals(RequestParameter("action")) &
"admin".equals(RequestParameter("user"))
/ counter = counter + 1;
SessionAttribute("isAdmin") = "true"
EXIT
F

Fig. 2. The Automaton Building Process

edges are propagated backward to the beginning of the true branch of the condition in line 11, and JMODEX substitutes the occurrences of variable *isAd* on these edges with the value assigned to it in line 12. The result is shown in Figure 2D.

Next, JMODEX arrives at the jump statement corresponding to the condition in line 11. This condition must be true in order to execute edges 3' or 3" and false to trigger the execution of edge 1 or 2. Thus, their guards are modified accordingly. Figure 2E shows that the guard of edge 2 has been simplified (e.g., if *action* is the *logout* string, *not("reset".equals(action))* is true and can be eliminated from the conjunction).

This process continues until all edges reach the beginning of the method, as seen in Figure 2F. All execution paths with the same *updates* are subsumed by a single edge in the model, with a guard that is the disjunction of the individual guards.

1) Loops: For methods with loops, we cannot determine all execution paths through the method. Consequently, we introduce an additional state/control point in the automaton, associated with the loop header. Moreover, an *update* to a relevant state variable may depend on other variables (including locals) updated within the loop. Thus, these variables are also relevant and their updates must be captured. The described algorithm is repeatedly applied for the loop until a fixed point is reached and the set of relevant variables no longer changes.

2) Infeasible paths: Some paths considered while building the automaton may in fact be non-executable. For example, in Listing 3, the path going through lines 8 and 10 as a result of satisfying the right operand of the disjunction in line 9 is infeasible: the *admin* variable cannot be simultaneously equal to "admin" (to execute line 8) and not be equal to it (to evaluate the right operand of the short-circuiting disjunction). JMODEX handles such cases by trying to identify contradictions in guard formulas. Additionally, JMODEX provides an API to integrate satisfiability checkers for logical formulas.

3) Limitations and Other Features: Currently, JMODEX cannot handle all *Java* constructs (e.g., polymorphic calls, exceptional paths, collection usage, etc.) or every possible *SQL* query. However, this does not hinder an initial evaluation of the tool to check the feasibility and usefulness of model extraction.

During this assessment, we identified that abstracting library or even application functions is critical to generating models that are compact enough to be handled by the model checkers. We have thus extended JMODEX with user-defined *specifiers*, which enable analysts to describe programmatically how certain methods (or even individual method invocations) should be analyzed during model extraction. The *specifier* mechanism can be used to express the semantics of the JSP framework methods in terms of the JMODEX meta-model, and could thus be used to extend JMODEX for analyzing applications built with other development technologies. We have also used specifiers to under-approximate the behavior of functions (e.g., ignore filtering or sanitizing of strings) when searching for logical flaws which are independent of these aspects.

B. ICONVERT – ASLAN++ *Generation*

In the second analysis step, JMODEX reduces the automaton/graph of each component using a structural analysis algorithm, obtaining for each automaton a *control tree* [7]. This tree is expressed in terms of ASLAN++ control statements and describes how statements are combined to produce the semantics of the automaton operations. We have defined translation schemas for each expression in the ISUMMARIZE meta-model; they are used to generate the ASLAN++ code for the guards and updates that fill the control tree. For example, the automaton in Figure 2F can be reduced to an ASLAN++ *select* statement that non-deterministically selects a transition with a true guard. Next, based on the translation schemas, JMODEX produces the code in lines 10 to 27 from Listing 1 (except the user-specified assertion), filling the *on* conditions with the edge guards and the statement bodies with the edge updates. Edges with no updates are eliminated if possible.

Finally, the ASLAN++ code of each component is inserted on a distinct branch of another *select* statement (e.g., line 8 in Listing 1). The purpose of this *select* is to enable a user (including an intruder) to access different components in every possible order she can imagine. Finally, everything is enclosed within the server infinite loop (e.g., line 7 in Listing 1).

IV. EVALUATION

For an initial evaluation of JMODEX we have used the *BookStore*[2] application in which we had manually detected a security vulnerability (later found to be already known[3]). The application has typical features for an on-line store (user registration, information updates, shopping cart, etc.) and it is developed using the JSP/Servlet technology. The security vulnerability is that a regular user can change the password of another user including that of the admin. In the following we describe the steps we applied in order to identify this vulnerability using the models extracted with JMODEX.

We start by extracting a model for the *Login* and *MyInfo* components, which perform user login and personal information updates. The corresponding JSPs have been converted in pure Java code using *Tomcat 6*'s translator, obtaining around 1500 lines of code. We have also used JMODEX facilities to simplify the application model, reducing its size to aid the verifier. Consequently, we have captured only those execution paths in which the request parameters exist (i.e., they are not *null* in the application), we have considered some sanitizing functions to be identities, etc. On a MacBook Pro computer (2.53 GHz Intel Core i5, 8GB RAM) running OS X Mountain Lion, JMODEX extracts in less than 30 seconds a model with around 140 lines of ASLan++ code. After initializing the model database and specifying the security requirement that a regular user should be able to modify only its entry in the database, we have verified the model using CL-ATSE. Limiting the maximum execution count of each transition to 2, CL-ATSE identified the attack in less than 10 seconds. Thus, we have shown that the model extracted using JMODEX is sufficient to highlight the initial vulnerability.

Next, we inserted an additional check in the *MyInfo* component to eliminate the vulnerability. After extracting the new model we re-ran the model checker, this time without finding any attack on the security property. Moreover, on the corrected application, the initial attack that changes the admin password fails. This confirms that JMODEX has correctly captured the behavior of the modified application in the new model.

V. RELATED WORK

Several other tools analyze application code to detect security vulnerabilities caused by faulty application logic. WALER [2] uses dynamic analysis to identify likely invariants and applies model checking over symbolic inputs to detect executions that may violate these invariants. WAPTEC [1] uses a constraint solver and dynamic analysis to identify executions triggered by user inputs that are rejected in the client side of the application but are incorrectly accepted by the server side.

In contrast, JMODEX only builds an application model; however, it is strongly linked to the model checkers used to verify its security properties (CL-ATSE [3] and SATMC [4]). It produces models expressible in their input specification language ASLAN++, and uses abstractions geared towards the features of this language and the checking algorithms.

BANDERA [8] has taken a similar approach of building models of Java programs that can be analyzed by independent model checkers. In comparison, JMODEX is not geared towards verification of concurrency and data structure properties, but produces models for model checkers that target security aspects. Consequently, it needs to incorporate security domain knowledge (e.g., about web applications and intruder actions), and generate models that can be efficiently checked.

VI. CONCLUSION

We have presented JMODEX, a tool to automatically extract behavioral models for web applications. The model is expressed in the ASLAN++ language and can be further used in conjunction with a model checker to verify security properties. Initial evaluation has showed that the tool can extract models from small but realistic web applications, and that the extracted models can be used to identify security vulnerabilities.

As future work, we plan to address the current limitations presented in Section III. A special focus is to provide more powerful abstraction/simplification mechanisms to address scalability issues raised by model checkers. An important ability would be to start from the security requirement to be checked and automatically abstract away during model construction all details that are irrelevant for the analysis goal.

ACKNOWLEDGMENT

This work has been supported by the European FP7-ICT-2009-5 project no. 257876 SPaCIoS — Secure Provision and Consumption in the Internet of Services

REFERENCES

[1] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan, "WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction," in *Proceedings, 18th ACM CCS*. ACM, 2011, pp. 575–586.

[2] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX Conference on Security*, 2010, pp. 143–160.

[3] M. Turuani, "The CL-Atse Protocol Analyser," in *Proceedings of RTA'06*, ser. LNCS, vol. 4098, 2006, pp. 277–286.

[4] A. Armando and L. Compagna, "SAT-based model checking for security protocol analysis," *Int. J. of Information Security*, vol. 7, pp. 3–32, 2008.

[5] M. Büchler, J. Oudinet, and A. Pretschner, "SPaCiTE – web application testing engine," in *Proceedings of Fifth ICST*, 2012, pp. 858–859.

[6] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Proceedings of Ninth FMCAD*. IEEE, 2009, pp. 25–32.

[7] S. S. Muchnick, *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: extracting finite-state models from Java source code," in *Proceedings of 22nd ICSE*. ACM, 2000, pp. 439–448.

[2]http://web.archive.org/web/20110430192101/http://gotocode.com/

[3]http://www.securityfocus.com/bid/49910/exploit