XSS Pattern for Attack Modeling in Testing

Josip Bozic
Institute for Software Technology
Graz University of Technology
A-8010 Graz, Austria
jbozic@ist.tugraz.at

Franz Wotawa
Institute for Software Technology
Graz University of Technology
A-8010 Graz, Austria
wotawa@ist.tugraz.at

***Abstract*—Security issues of web applications are still a current topic of interest especially when considering the consequences of unintended behaviour. Such services might handle sensitive data about several thousands or millions of users. Hence, exploiting services or other undesired effects that cause harm on users has to be avoided. Therefore, for software developers of such applications one of the major tasks in providing security is to embed testing methodologies into the software development cycle, thus minimizing the subsequent damage resulting in debugging and time intensive upgrading. Model-based testing evolved as one of the methodologies which offer several theoretical and practical approaches in testing the system under test (SUT) that combine several input generation strategies like mutation testing, using of concrete and symbolic execution etc. by putting the emphasis on specification of the model of an application. In this work we propose an approach that makes use of an attack pattern model in form of a UML state machine for test case generation and execution. The paper also discusses the current implementation of our attack pattern testing tool using a XSS attack pattern and demonstrates the execution in a case study.**

***Index Terms*—Attack pattern model, cross-site scripting, model-based testing, security testing.**

I. INTRODUCTION

With higher complexity of modern days web applications, already known security breaching methods become more sophisticated. It remains the task of the developer to adapt on new circumstances by considering new detection and prevention mechanisms. A previous report [1] shows that the most common software exploitation methods are still cross-site scripting (XSS) and SQL injection (SQLI) despite the fact that several protection mechanisms are already discussed and implemented.

The current research from the area of model-based testing offers several methods and solutions in order of how to formalize and implement testing techniques that are able to detect potential security leaks in programs. These methods differ accordingly to the initial problem statement. For example, if the source code of the system under test (SUT) is unknown, black-box techniques like fuzz testing are the first choice. Fuzz testing is an optimizing random test case generation method that also makes use of underlying models like communication process models [2]. On the other hand, if a developer wants to test his or her own implementation having complete insights of the source code, white-box testing methods may be applied [3].

In this paper we focus on another direction and concentrate on specifying effective testing mechanisms in the domain of security that can be easily integrated into today's software development processes. Actually, the most demanding and promising task is the complete automation of the testing process, i.e. ensuring the immunity of an application and liberating the tester from time-consuming manual testing work.

In order to make automation possible, all attack vector information must be gathered and structured in one single representation. For this sake we propose the use of attack patterns, i.e. methods which describe all pre- and postconditions, attack steps as well as the expected implications for an attack to be carried out successfully. We formalize the pattern using the UML state machine modeling language [4] and integrate the whole testing system into the Eclipse development environment.

In our approach an attack pattern model is executed against the application, reporting a positive or negative feedback. In our running example, we specify a XSS attack pattern using UML state machines. Each execution step can be carried out accordingly to this model, branching through all conditional paths inside its structure. All operations are defined as parts of transitions and states and are called whenever activating one of the states when traversing through the model. If a path does not lead to vulnerability detection, another state is entered, thereby generating another input, which represents a new test case and is executed against the SUT again. Because test case generation depends solely upon predefined methods, a tester may add manually additional methods, thus extending the model whenever needed. When considering that the tester already knows the source code of the application, he or she is asked to add manually an expected postcondition inside the testing application as trigger events.

II. RELATED WORK

There are several papers dealing with attack modeling, automatic test generation and execution in the context of UML-based and security testing.

Kim and colleagues [5] discuss the usage of UML state machines for class testing. In their approach, a set of coverage criteria is proposed according to the control and data flow in the model and the test cases are generated from the diagram by satisfying these criteria. The authors describe a method on how to transform the state machine into an extended finite state machine (EFSM) that are in turn transformed into a flow graph so common data flow analysis methods and a coverage criteria can be applied in order to generate test cases as a set of paths by using all data from the model.

Kiezun et al. [6] propose a technique for automatic test case generation for XSS and SQLI for testing of PHP applications by using concrete and symbolic execution. They take into consideration persistent cross-site scripting. For this task to be realized, they implemented the tool ARDILLA, which is able to use any input generator. The tool generates inputs and executes these inputs against the SUT. Hence, path constraints are generated and the test case generator can negate one of the constraints, thus eventually producing a solution for the new constraint system. ARDILLA tracks all user inputs through the application in order to find out whether it can lead to a security breach.

Tappenden et al. [7] concentrate on security issues during the development and testing processes of web applications using agile methodologies, which includes the modeling of security requirements, employing a highly testable architecture and describing and execution of automated tests by using the testing framework *HTTPUnit* [8]. Within this testing framework security issues can be formalized and executed.

Offutt et al. [9] describe how to bypass client-side validation of user input data by generating tests, which solve all predefined input constraints. According to their paper, first constraints on user inputs are detected. Then, several test cases are generated by solving the identified constraints, thus trying to bypass value, parameter and control flow definitions.

The authors from [10] describe three different model-based testing approaches and propose the usage of models of attack patterns for automatic test case generation and execution. In this paper we concentrate on that suggestion and elaborate our method in the case study.

Other works that put more emphasis on the aspect of modeling by using activity diagrams include [11] and [12]. Furthermore, [13] deals with the application of state machines. Also, there are several other works dealing with different approaches to model-based testing [14], [15] and [16]. These papers put a greater emphasis on fuzz testing approaches and black-box testing techniques.

III. Case Study

Cross-site scripting is defined as a method to force a website to execute injected malicious code. Accordingly to [17], XSS attacks are commonly categorized as *non-persistent (or reflected; "type-1")*, *persistent (or stored; "type-2")* and *DOM-based (sometimes "type-0")*.

Starting with the definition from Moore et al. [18] defining an attack pattern as a representation of an attack that occurs under certain circumstances, each pattern must have the following attributes: goal of the attack, some preconditions, all attack steps, and the postconditions.

For type-1 and type-2 of attack we implemented a small PHP application, which are in fact a login form and a registration form and include only the most important elements in order to demonstrate our testing approach. The SUT is deployed on a Apache Server and compromise a common MySQL database. The first application for non-persistent XSS has a GUI and offers the possibility to log into the application by submitting a username and a password into separate fields. Its source code contains several elements, attributes and data which will be automatically extracted by our implementation and are needed to craft new HTTP responses to be submitted.

The application for stored XSS exploitation testing is very similar to the one above. The only difference is that it stores the data into the database and does not return any submitted data so actually it does not function for logging purposes. Because both applications use the same database, data from a new registered user is retrieved via the type-1 XSS program by submitting the correct credentials.

Regarding the modeling of the attack pattern we decided to adapt a state machine diagram to our needs because of its widespread usage and offered modeling possibilities. For the sake of simplicity of our implementation, we use the open-source tool-kit YAKINDU Statechart Tools[1]. The tool allows specifying interfaces, variables and operations in an editor which is offered beside the modeling part which is an important feature because all data definitions and methods for test execution and generation will be specified in form of one of these data types.

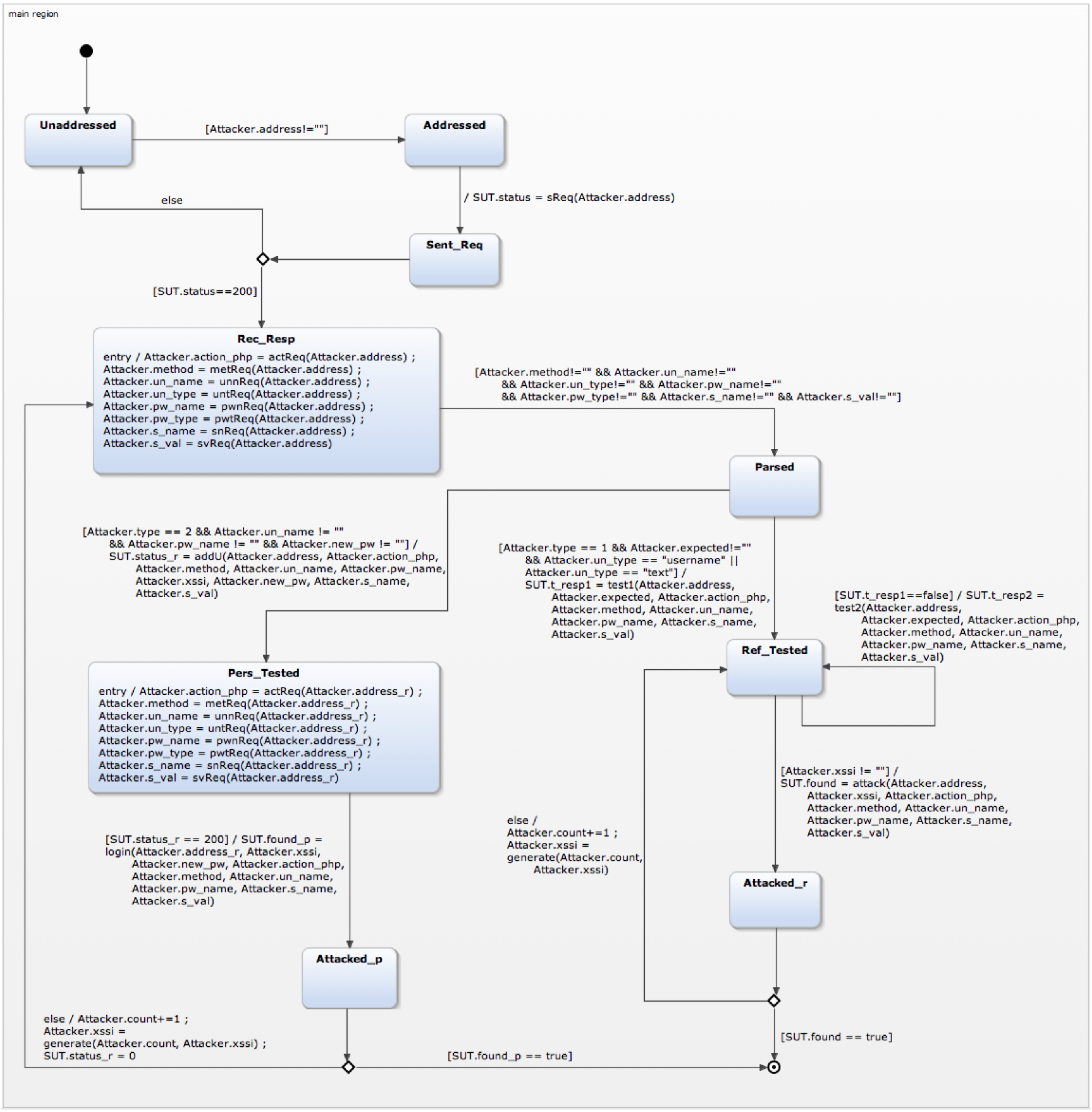
One important fact of our approach is that the model covers just the attack pattern and not the inner structure of the SUT. In contrast to other model-based testing techniques that use a correct version of the application and compare the output of the real program with the expected one derived from the model. We do not test the functionality of the SUT at all but concentrate entirely on the test case execution in order to identify a vulnerability.

All communication between the attack pattern model execution and the application takes place by bypassing the browser, thus evading all browser specific XSS filters. From the practical point of view, we avoid these defenses by implementing a HTTPClient[2] in Java and use it to create, send and read HTTP messages. After reading a response, we parse the obtained HTML or XML code using a DOM parser. By using these methods, the whole document is segmented into elements, attributes and values so the entire response message can be extracted by searching for key characters or words. In this way an encountered <*script*> element can be detected and so it becomes obvious that the application is vulnerable to script injections. But if the inserted script is returned in form of a normal string value, the XSS attack was in vain. The whole attack model for both type-1 and type-2 XSS attacks adapted for testing our SUT is depicted in Figure 1.

We define two interfaces, respectively one for the *Attacker* and the other one for the *SUT* and other initial variable values, e.g. the URL address of the application, initial XSS input string etc. which represent preconditions of the execution

[1]http://statecharts.org/

[2]http://hc.apache.org/httpcomponents-client-ga/



main region
Unaddressed
[Attacker.address!=""]
Addressed
/ SUT.status = sReq(Attacker.address)
Sent_Req
else
[SUT.status==200]
Rec_Resp
entry / Attacker.action_php = actReq(Attacker.address) ;
Attacker.method = metReq(Attacker.address) ;
Attacker.un_name = unnReq(Attacker.address) ;
Attacker.un_type = untReq(Attacker.address) ;
Attacker.pw_name = pwnReq(Attacker.address) ;
Attacker.pw_type = pwtReq(Attacker.address) ;
Attacker.s_name = snReq(Attacker.address) ;
Attacker.s_val = svReq(Attacker.address)
[Attacker.method!="" && Attacker.un_name!="" && Attacker.un_type!="" && Attacker.pw_name!="" && Attacker.pw_type!="" && Attacker.s_name!="" && Attacker.s_val!=""]
Parsed
[Attacker.type == 2 && Attacker.un_name != "" && Attacker.pw_name != "" && Attacker.new_pw != ""] / SUT.status_r = addU(Attacker.address, Attacker.action_php, Attacker.method, Attacker.un_name, Attacker.pw_name, Attacker.xssi, Attacker.new_pw, Attacker.s_name, Attacker.s_val)
[Attacker.type == 1 && Attacker.expected!="" && Attacker.un_type == "username" || Attacker.un_type == "text"] / SUT.t_resp1 = test1(Attacker.address, Attacker.expected, Attacker.action_php, Attacker.method, Attacker.un_name, Attacker.pw_name, Attacker.s_name, Attacker.s_val)
[SUT.t_resp1==false] / SUT.t_resp2 = test2(Attacker.address, Attacker.expected, Attacker.action_php, Attacker.method, Attacker.un_name, Attacker.pw_name, Attacker.s_name, Attacker.s_val)
Ref_Tested
Pers_Tested
entry / Attacker.action_php = actReq(Attacker.address_r) ;
Attacker.method = metReq(Attacker.address_r) ;
Attacker.un_name = unnReq(Attacker.address_r) ;
Attacker.un_type = untReq(Attacker.address_r) ;
Attacker.pw_name = pwnReq(Attacker.address_r) ;
Attacker.pw_type = pwtReq(Attacker.address_r) ;
Attacker.s_name = snReq(Attacker.address_r) ;
Attacker.s_val = svReq(Attacker.address_r)
[Attacker.xssi != ""] / SUT.found = attack(Attacker.address, Attacker.xssi, Attacker.action_php, Attacker.method, Attacker.un_name, Attacker.pw_name, Attacker.s_name, Attacker.s_val)
else / Attacker.count+=1 ; Attacker.xssi = generate(Attacker.count, Attacker.xssi)
[SUT.status_r == 200] / SUT.found_p = login(Attacker.address_r, Attacker.xssi, Attacker.new_pw, Attacker.action_php, Attacker.method, Attacker.un_name, Attacker.pw_name, Attacker.s_name, Attacker.s_val)
Attacked_r
Attacked_p
else / Attacker.count+=1 ; Attacker.xssi = generate(Attacker.count, Attacker.xssi) ; SUT.status_r = 0
[SUT.found_p == true]
[SUT.found == true]

Fig. 1. An UML based XSS attack model

of the attack model and are obligatory. All the values for other parameters will be set during the execution as results of method calls and will guide the processing further by being parts of guards or actions.

In our case study, the expected test outputs are defined within method calls and Boolean variables and so for a test case to be successful, the variable *found* is expected to be true.

Stored XSS needs a few steps more because first a data has to be stored before analyzing the feedback. For this sake, at least one of the inputs must have the script input as its content. Then, a logging is carried out with the same credentials, i.e. using the whole script input for the username via the SUT for type-1. Here the response is parsed again and if the inserted script element is found, the test is successful.

The next important topic of using attack pattern models for test case generation is the test generation process itself. If a test case fails, another input with a different syntax is returned from a generation method in real-time, which is submitted against the SUT. The task of the tester is to manually implement these input generation sub methods by taking into consideration all known state-of-the-art XSS filtering techniques and special cases. The whole process continues as long as the expected affirmative value is not received or no more test cases are generated.

Our case studies are adapted for an application, which has only input fields specified for data submission. During execution, this SUT specific data must be extracted and used in the model. In such way, the program finds out where inputs can be submitted so this is the actual exploitation starting point.

The attack pattern execution from the state machine starts in the initial state. After the communication with the SUT is established, several methods are called in order to get all parameter values for a valid HTTP request. The variables collect information about the HTTP method, URI and input fields from the source code by parsing HTTP responses that are generated after sending requests during the methods execution. This is the switching point between the two different attacks. The path below from *Parsed* leads to reflected detection attempts and the path to the left towards the stored attack.

For reflected vulnerability, we try to find out if the SUT returns a user input within the body of a response. The first attack is carried out by submitting the initial script into the same field as indicated to be sent back to the tester. When done, the HTTP response is read by parsing the DOM of its body and if the <*script*> element is found, this indicates a potential non-persistent XSS vulnerability. If the test result is negative, the input generation transition is taken and a new attack input is submitted against the application. This process will continue as long as a successful attack is not carried out or until no more test cases are generated, thus remaining stuck in the model.

For the persistent part of the attack model, a new user is saved into the database but here the attack script will be inserted instead of a casual username. Then, the login site is called with all its parameters in order to authenticate with the previously submitted values. If all request data is retrieved, the message is sent and from here the procedure is the same as with reflected XSS, meaning that if the script is read as a DOM element, the attack reports success and enters the final state. Otherwise, the generator transition is entered and the process returns a little further back and enters a new string into the registration form again. The generator transition is the same for both attack types because only one path is taken so there won't be any value conflicts.

IV. Conclusion and Future Work

In this work we presented a method which introduces the formal definition of an attack pattern for cross-site scripting in form of a UML state machine. Further we executed this attack model against two web applications, respectively one for type-1 and type-2 vulnerabilities. The most important fact is that the model, once specified, executes fully automatically and reports its status.

The same approach may be also redefined, for example by adapting a different test generation technique or by changing the method of how vulnerabilities are detected. But nevertheless, similar attack patterns can be defined as proposed in this paper and can also be extended after the model is already set up.

It remains open to adapt this approach on large real-world applications, especially by adapting it against sophisticated defense mechanisms.

Acknowledgement

The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 832185 (MOdel-Based SEcurity Testing In Practice) and ITEA-2 (DIAMONDS).

References

[1] J. Williams and D. Wichers, "OWASP Top 10 2010," 2010, https://www.owasp.org/index.php/Top_10_2010-Main.

[2] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, USA, 2008.

[3] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI*, 2008, pp. 206–215.

[4] "OMG Unified Modeling Language Infrastructure Version 2.4.1," 2011, http://www.omg.org/spec/UML/2.4.1/Infrastructure.

[5] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha, "Test cases generation from uml state diagrams," in *IEEE Proceedings-Software, 146(4)*, 1999, pp. 187–192.

[6] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'09)*, 2009.

[7] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, "Agile security testing of web-based systems via httpunit," in *Proceedings of the Agile Development Conference (ADC'05)*, 2005, pp. 29–38.

[8] R. Gold, "Httpunit," 2003, http://httpunit.sourceforge.net/.

[9] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass testing of web applications," in *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, 2004.

[10] J. Bozic and F. Wotawa, "Model-based testing - from safety to security," in *Proceedings of the 9th Workshop on Systems Testing and Validation (STV'12)*, October 2012, pp. 9–16.

[11] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, and X. Li, "Uml activity diagram-based automatic test case generation for java programs," in *The Computer Journal*, 2007.

[12] A. Heinecke, T. Brueckmann, T. Griebe, and V. Gruhn, "Generating test plans for acceptance tests from uml activity diagrams," in *Proceedings of the 17th International Conference on Engineering Computer-Based Systems, IEEE*, 2010.

[13] A. Beer, S. Mohacsi, and C. Stary, "IDATG: An Open Tool for Automated Testing of Interactive Software," in *Proceedings of the COMPSAC'98 - 22nd International Computer Software and Applications Conference*, 1998.

[14] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-based security testing," in *Proceedings of the Model-Based Testing Workshop at ETAPS 2012. EPTCS*, 2012, pp. 1–12.

[15] S. Rawat and L. Mounier, "Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, March 2011, pp. 531–533.

[16] A. Takanen, "Fuzzing: the past, the present and the future," in *SSTIC'09*, 2009.

[17] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov, *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2007.

[18] A. P. Moore, R. J. Ellison, and R. Linger, "Attack Modeling for Information Security and Survivability," in *Technical Note CMU/SEI-2001-TN-001*, March 2001.