2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

SPaCiTE - Web Application Testing Engine

Matthias Büchler
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
buechler@kit.edu

Johan Oudinet
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
oudinet@kit.edu

Alexander Pretschner
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
pretschner@kit.edu

***Abstract*—Web applications and web services enjoy an ever-increasing popularity. Such applications have to face a variety of sophisticated and subtle attacks. The difficulty of identifying respective vulnerabilities steadily increases with the complexity of applications. Moreover, the art of penetration testing predominantly depends on the skills of highly trained test experts. The difficulty to test web applications hence represents a daunting challenge to their developers. As a step towards improving security analyses, model checking has, at the model level, been found capable of identifying complex attacks and thus moving security analyses towards a push-button technology. In order to bridge the gap with actual systems, we present SPaCiTE. This tool relies on a dedicated model-checker for security analyses that generates potential attacks with regard to common vulnerabilities in web applications. Then, it semi-automatically runs those attacks on the System Under Validation (SUV) and reports which vulnerabilities were successfully exploited. We applied SPaCiTE to Role-Based-Access-Control (RBAC) and Cross-Site Scripting (XSS) lessons of WebGoat, an insecure web application maintained by OWASP. The tool successfully reproduced RBAC and XSS attacks.**

I. Introduction

Since the emergence of model-checkers dedicated to security goals [3], developing secure specifications is easier. However, testing if the implementation of such specifications is also secure remains a difficult task. Existing penetration testing tools for web applications either require access to the source code [8] or suffer from missing a lot of potential interactions with the user; Doupé et al. [7] evaluated the weaknesses of such penetration scanners. Even though the idea of exploiting the availability of a secure model to test an implementation is not new [2, 1, 6], SPaCiTE focuses on testing web applications and therefore provide an even more practicable tool for testing the security of web applications. First, it allows to abstract away the protocol layer (HTTP) from the model. Thus, modeling web applications can be done at the browser level, which both simplify the modeling process and include the browser as part of the testing process. Second, SPaCiTE relies on a library of known vulnerabilities and attacks to select interesting test cases for web applications. Indeed, the tool successfully reproduced RBAC and XSS attacks in four lessons of WebGoat[1].

The methodology behind the tool is described in detail in [5], and a demonstration video of the tool is available on-line[2]. In this paper, we first describe the overall workflow of the tool and then provide details on the Test Execution Engine (TEE) architecture. Finally, we report in Section III the results from executing the tool on several WebGoat lessons.

[1]Webgoat: www.owasp.org/index.php/Category:OWASP_WebGoat_Project
[2]SPaCiTE demo: zvi.ipd.kit.edu/26_500.php

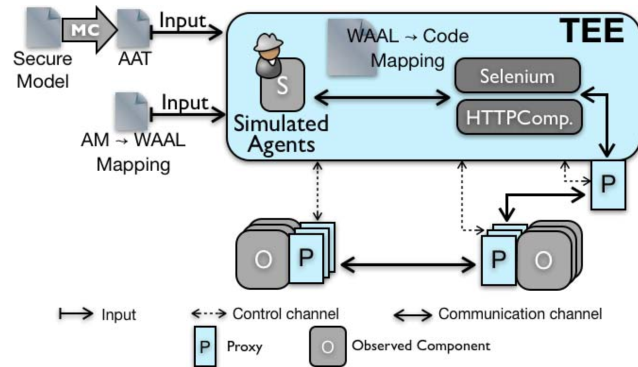

Fig. 1. SPaCiTE workflow and TEE architecture

II. The SPaCiTE Tool

A. *Workflow*

The SPaCiTE tool (Figure 1) takes as input a secure model described in the AVANTSSAR Specification Language (ASLan++) [4]. Even though ASLan++ was created for security protocols, the language can naturally be used to model web applications as well. In particular the definition of access control policies in web application is simplified by the availability of horn clauses. The model is described in terms of abstract messages exchanged by different web application components. These messages describe the interaction between components at a high level (e.g., `login`, `viewProfile`, `updateProfile`), ignoring details about underlying protocols.

First, SPaCiTE injects some known vulnerabilities into the secure model such that a model checker may report Abstract Attack Traces (AATs) that exploit these vulnerabilities. An AAT is a sequence of abstract messages together with their sender and receiver. Then, using a mapping given as input to the tool, every AAT message is mapped to browser actions that generate and verify these messages. An intermediate language, called Web Application Abstract Language (WAAL), has been developed to simplify the translation of abstract messages to browser actions. Next, WAAL actions are mapped to executable code by an internal, framework-specific mapping. Finally this concrete test case is executed against the SUV by a TEE. The last step is described in detail in the following section.

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.187

IEEE computer society

B. Test Execution Engine

First, SUV components are split into two distinct sets: simulated and observed components. The simulated agents are part of the TEE, which is responsible for emitting the messages they are supposed to send according to the test case. Observed components run in their normal environment except that their communication channels are monitored by proxies. These proxies allow the TEE to observe messages exchanged between observed components. An additional proxy is put in front of the TEE to intercept sent messages when it generates additional messages that are not part of the test case; such extra messages are for the test expert when the TEE requests his intervention.

Then, the TEE translates WAAL actions performed by simulated agents into pieces of code executable within the Selenium framework[3] and/or Apache HTTPComponents[4]. Such code also contains some recovery actions that are executed when a browser action cannot be performed (e.g., selecting an item from a list that is missing, clicking on a hidden button).

If the code executed by Selenium fails, it first tries to execute some prepending actions (e.g., reconfiguration of the tool to use credentials) and then executes the failed action again, or some alternative actions (e.g., triggering the desired event by a different action). These automatic recovery actions depend on the occurred failure. If neither prepending nor alternative actions can recover from the failure, the TEE switches to the HTTP level and makes use of the HTTPComponents framework. In this case, the TEE may require the help of a test expert to translate a sequence of browser actions into the corresponding HTTP request. To simplify the task of the test expert, the TEE will provide HTTP requests that correspond to similar actions. For example, if an item is missing in a list, the TEE will show the HTTP request that corresponds to selecting another item from the same list. Once the test expert provided the needed HTTP request (i.e., the failed Selenium action is successfully executed by the HTTPComponents framework), the test case is resumed in the Selenium framework. Shared information (e.g., cookies) between the frameworks is automatically transferred.

III. Application to WebGoat

To evaluate the tool, we considered four WebGoat lessons from the RBAC and XSS domains. For all these lessons, the SUV is split into the web server for the observed component and the clients for the simulated agents, which access to the server through a web browser. Our tool successfully instantiated and executed attack traces reported by the model checker.

For the RBAC domain, the models describe a web application where a user can view and/or delete user profiles only if he is authorized to do so with respect to the RBAC system. After mutation operators seeded vulnerabilities that deactivate server-side authorization checks, the model checker reported AATs that show how a user can view or delete unauthorized profiles. SPaCiTE was able to automatically execute all AATs messages but the last one. To also successfully execute the last message, the presentation layer must be bypassed (i.e., by moving to the HTTPComponents framework). To help the test expert in providing the mapping of the last message to an HTTP message, SPaCiTE shows him some HTTP messages that result from executing similar actions. For example, in the "Bypass Data Layer Access Control" lesson, SPaCiTE shows HTTP requests for viewing other profiles. For the "Bypass Business Layer Access Control" lesson, the HTML button to delete a profile is missing on the webpage and thus SPaCiTE shows HTTP requests generated by other buttons on the same page.

For the XSS domain, the models describe a web application where a user can update profiles and search for a profile via a query engine. In the secure models, every action sanitizes user inputs. After tagging some actions as non-correctly sanitizing, the model checker reports AATs showing a specific sequence of messages to exploit a potential XSS vulnerability. During the execution of these AATs, the non-correctly sanitized input of the vulnerable request is replaced by a malicious JavaScript code. Then, a verification code is executed at the place where this JavaScript code should eventuate. Both the malicious JavaScript code and the corresponding verification code come from a library SPaCiTE has access to. The two AATs for XSS were automatically be executed by the tool.

To sum up, all reported attack traces has been successfully reproduced. Future versions of the tool will be implemented as a web service to bring SPaCiTE to the SUV such that the test environment can stay locally.

Acknowledgment

This work was partially supported by the FP7-ICT-2009-5 Project no. 257876, "Secure Provision and Consumption in the Internet of Services" (http://www.spacios.eu).

References

[1] P. Ammann, W. Ding, and D. Xu, "Using a model checker to test safety properties," in *ICECCS*, 2001, pp. 212–221.

[2] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *ICFEM*, 1998, pp. 46–54.

[3] A. Armando and L. Compagna, "Sat-based model-checking for security protocols analysis," in *IJISEC*, 2008, pp. 3–32.

[4] AVANTSSAR, "Deliverable 2.3 (update): ASLan++ specification and tutorial," 2011, http://www.avantssar.eu.

[5] M. Büchler, J. Oudinet, and A. Pretschner, "Semi-automatic security testing of web applications from a secure model," KIT, Tech. Rep. 1000025844, 2012.

[6] F. Dadeau, P.-C. Héam, and R. Kheddam, "Mutation-based test generation from security protocols in HLPSL," in *ICST*, 2011, pp. 240–248.

[7] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: an analysis of black-box web vulnerability scanners," in *DIMVA*, 2010, pp. 111–131.

[8] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *ICSE*, 2009, pp. 199–209.

[3] Selenium: http://seleniumhq.org/

[4] Apache HTTPComponents: http://hc.apache.org/