Model Inference and Security Testing in the SPaCIoS Project

Matthias Büchler[†], Karim Hossen[*], Petru Florin Mihancea[‡], Marius Minea[‡], Roland Groz[*], Catherine Oriat[*]

[*]LIG Lab, University of Grenoble
Grenoble F-38402, France
{hossen, groz, oriat}@imag.fr

[†]Technische Universität München
Germany
buechler@cs.tum.edu

[‡]Institute e-Austria Timișoara, Romania
Politehnica University of Timișoara, Romania
{petru.mihancea,marius}@cs.upt.ro

***Abstract*—The SPaCIoS project has as goal the validation and testing of security properties of services and web applications. It proposes a methodology and tool collection centered around models described in a dedicated specification language, supporting model inference, mutation-based testing, and model checking. The project has developed two approaches to reverse engineer models from implementations. One is based on remote interaction (typically through an HTTP connection) to observe the runtime behaviour and infer a model in black-box mode. The other is based on analysis of application code when available. This paper presents the reverse engineering parts of the project, along with an illustration of how vulnerabilities can be found with various SPaCIoS tool components on a typical security benchmark.**

***Index Terms*—Control Flow Inference, Data-Flow Inference, Security, Web Application, Reverse-Engineering,**

I. SPaCIoS Project

In the Internet of Services, the way applications are designed, implemented, deployed and consumed has changed. An application is now composed of several components that are distributed over the network, aggregated and consumed at runtime a in demand-driven way. The goal of the SPaCIoS project[1] (Secure Provision and Consumption in the Internet of Services) [1] is to build an integrated platform to discover security flaws in such systems using a model-based approach. SPaCIoS continues the work of the AVANTSSAR [2] project which proposed an automated method for validation of trust and security of services.

Figure 1 shows possible workflows of the SPaCIoS tool. Since both model checking and security testing in the project take a model as input, reverse engineering a suitable model of the application under scrutiny is essential. We have developed two approaches for this purpose: SIMPA (a recursive acronym for "SIMPA Infers Model Pretty Automatically"), a black-box model inference component which uses the interaction with the application to incrementally build the corresponding model, and JMODEX, a white-box model extraction component which works from the source code when it is available. Both of them generate a model in the high-level formal language ASLan++ [3], the modelling language of SPaCIoS.

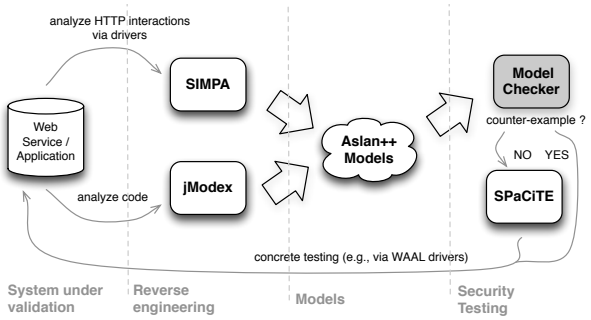

Fig. 1. Model Extraction and Security Validation in SPaCIoS

Models are then augmented with security goals and properties, and a model checker generates abstract traces which may violate the properties: the potential attacks. For vulnerability detection, we have also developed SPaCiTE, a property- and vulnerability-driven test case generator and execution engine. It follows a mutation-based approach that creates potential attack traces from a secure model. Finally, the attack traces are executed on the SUT (System Under Test) using the test execution engine to confirm the attacks.

In the following sections, we describe the components presented above and show current results obtained by applying some of these tools on a typical security benchmark.

II. Black-box Reverse Engineering of Models

In order to apply the vulnerability detection techniques, the security analyst needs to provide a formal model of the application in the ASLan++ language. SIMPA is the black-box component that reverse-engineers a model from the implementation of the System under Validation (SUV), without any access to the code, even at assembly level.

A. Interfacing to web applications

Both inferring a model and testing an abstract trace on the system require bridging the gap between abstract modeling and concrete implementation levels. We have therefore developed

This work was supported by the FP7-ICT-2009-5 Project no. 257876, "SPaCIoS: Secure Provision and Consumption in the Internet of Services"

[1]SPaCIoS is a 40-month STREP project that started on October 1, 2010, with a total budget of 5.595 M€, of which 3.61 M€ EU contribution. The consortium is formed of 8 academic and industrial research teams: University of Verona, Italy; ETH Zurich, Switzerland; University of Grenoble, France; University of Genova, Italy; Institute e-Austria Timișoara, Romania; Technische Universität München, Germany; SAP AG, France; Siemens AG, Germany. The detailed list of participants is available on the project website, www.spacios.eu

a method to generate a test driver for a web application using a crawler, with minimal effort for the security tester.

First, we define the input and output from the model inference point of view. An input I of a web application is a tuple (M, A, P): M is the request method, e.g, GET, A is the address of the page handling the request and P is a finite set of pairs (name, value).

The inference algorithm uses the output of the application to distinguish between states. As a web application uses HTML to represent a page, we define an output as an HTML page with an abstraction of the content. Only the structure and the inputs of the page are considered. In this way, pages with different source code can represent the same output with different parameter values. To extract the position of the parameters in the pages, each time we get an input I leading to an output O, we fuzz the parameter values of I to generate multiple versions of the page, then we compare the source code of these pages to detect the position of the different parameters.

The crawler's goal is to discover as many HTML pages as possible, maximizing the number of different inputs and outputs. As the page visiting order is important, we reset the application before applying each input sequence to prevent previous requests from changing the application state. Crawling stops when all discovered inputs have been processed. In web applications, millions of pages may each be associated with one parameter value, e.g., a gallery of pictures represented by an ID. To avoid crawling them all, we defined rules like stopping the crawler when two pages differing in only one input parameter (ID) lead to the same output.

We also need to provide the crawler with some critical values that cannot be found, guessed or generated automatically by the crawler, e.g., credentials needed to reach all pages. This step is the only manual part of the test driver generation.

After crawling, an XML file is created containing input and output symbols, their parameters and the set of different outputs. From this file, the test driver is instantiated, providing functions to convert abstract inputs into concrete HTTP requests, and concrete HTTP responses into abstract outputs.

The inference algorithm and the crawler are implemented in the same tool named SIMPA. A module to load and use the driver is available from the SPaCIoS tool but also for third-party software. More details about the method are given in [4].

B. Overview of the inference algorithm

SIMPA implements an improved version of the inference algorithm from [5]. We adapted the learned model to the SPaCIoS context and especially to web applications by considering an EFSM (Extended Finite State Machine) as the model: an FSM with parameterized transitions, guards, and a function for each output parameter. Basically, the algorithm explores each input of the system for each discovered state and records the associated output to be able to detect new distinct states.

The values of the parameters are also stored to be processed by data mining algorithms in order to produce decision trees and build compact guards and better output functions.

C. Non-deterministic parameter detection

Whether for authentication or to prevent replay attacks, most applications use non-deterministic values, such as session-ID. Our algorithm can handle these non-deterministic values (*ndv*), which are critical for security. To detect them, we consider the system as deterministic; in this way, in the same state, the same input with the same parameters should produce the same output with the same parameters. Otherwise, we consider the different parameters as *ndv*. Then, their current value will be used for further tests to discover potential new behaviours.

D. Data mining

For building guards and output functions we use two different algorithms : the J48 decision tree learning algorithm [6] and M5P [7] for string and number parameters, respectively. This step is important for two reasons: it allows us to build more realistic and compact guards, and to identify reflected values which are critical to detect XSS (cross-site scripting).

III. White-Box Reverse Engineering

A formal model can also be reverse engineered from the application code, if available. The JMODEX tool [8] implements this white-box modeling approach for JSP web applications.

1) Extracting behavioral automata: JMODEX first builds an *extended finite state machine* for each component (servlet) of the target system. A node in this automaton represents a control point in the component code (e.g., entry / exit point). An edge subsumes all component execution paths that produce the same *updates*, or state changes, and also has a *guard* formula representing the conditions that trigger its execution.

At present, JMODEX can analyze servlet-based applications. It captures updates to session attributes, the usage of request parameter values, etc. JMODEX can also model the database of an application and analyze a subset of SQL, capturing database updates, the usage of particular database values, etc.

2) ASLan++conversion: As second step, JMODEX converts the automata of all application components into a formal model in the ASLan++ language. The automaton control flow is expressed using ASLan++ statements, and translation schemas are applied for each expression type that can arise in an application. For instance, since ASLan++ does not support arithmetic, integer operations are modeled by uninterpreted functions with properties defined as Horn clauses.

JMODEX can be used as an Eclipse plugin and enables a user to intervene in the model extraction process in order to reduce the size of the resulting model (e.g., specify under-approximations of the system behavior, abstract away parts of the investigated program, etc.). In a similar manner, a user can describe the semantics of library methods which are part of other web development frameworks. Thus, a user may extend the usage of our tool to other development technologies.

Currently, JMODEX does not handle all Java constructs (e.g., polymorphic calls, exceptional paths). Nevertheless, an extracted model has been already used successfully [8] to detect a known access control vulnerability in a web application with about 1500 lines of code, resulting in a model of 140 lines.

IV. Mutation-based Vulnerability Detection

The mutation-based vulnerability detection approach starts with secure ASLan++ models. A secure model is by definition one in which a model checker cannot find any trace that violates the defined security properties. If the initial model already violates a high level security property, the model checker returns a counter-example trace that could be interpreted as an abstract test case. Such an insecure model is eventually improved by the modeler until it does not violate any security property anymore. The approach addresses the issue how to proceed from such a model, since it cannot be used directly for test case generation. Nevertheless, semantic mutation operators can be applied to such secure models in order to inject common vulnerabilities and check whether they lead to traces that violate security properties. This approach is implemented in SPaCiTE, a tool that generates and executes "interesting" test cases for web applications [9]. SPaCiTE is integrated into the SPaCIoS tool as an Eclipse plugin, and uses model checkers developed in the AVANTSSAR [2] project.

Semantic Mutation Operators

SPaCiTE generates test cases from a secure model by applying semantic mutation operators to it. These mutation operators capture source code vulnerabilities that a software developer might mistakenly cause during coding. By applying a mutation operator to a model, the abstraction of the represented vulnerability is injected into the model. This step requires that SPaCiTE understands the semantics of different parts of the model. The semantics is provided in the form of annotations. According to the presence of different annotations, SPaCiTE applies a set of appropriate mutation operators on the annotated secure model. At present, SPaCiTE provides mutation operators for SQL injections, Reflected and Stored XSS, and Access Control vulnerabilities. Examples of corresponding source code level vulnerabilities are missing or incomplete permission checks, missing or incomplete input sanitizations, and configurations that might violate high-level access control properties.

Each of these categories consists of a set of mutation operators, since a source code vulnerability can be represented in different ways. The decision which concrete operators are applied is determined by modeling details of the secure model.

Applying mutation operators generates a set of mutated models which is given to a model checker to look for counter-examples. Such counter-examples are considered "interesting" test cases because they violate a specified security property of the web application by exploiting an injected vulnerability.

V. Concretization from Model: WAAL

Whenever a model checker reports a counter-example, it is at the same level as the input model. Therefore it needs to be concretized to be executed on the SUV (Figure 1).

An abstract attack trace consists of a sequence of abstract messages that are exchanged between different components. Depending on which agents are simulated and which are executed in the original form, the set of abstract messages is split into messages that need to be generated (G-messages) and messages that need to be verified (V-messages). Such an abstract attack trace can be instantiated using a 2-step mapping that involves an intermediate language called Web Application Abstract Language (WAAL). The central idea behind WAAL is that a sequence of abstract messages (output of the model checker) is mapped to a sequence of browser actions (provided by the WAAL language). The language is composed of two types of actions: Generating Actions (GA) and Verifying Actions (VA). The former aim at describing how to generate an abstract message by executing actions in a browser and the latter how to verify an abstract message.

In the first step, the security analyst provides a mapping with the following characteristics: Upon executing the sequence of browser actions, protocol-level messages are generated that are an instantiation of G-messages and browser actions are performed to check for V-messages. As a simplified example, the abstract action "login(tom,pwd)" may be mapped to inputText(tom), inputText(pwd), clickButton(Sign In), whereas an abstract message like "list_of_profiles" is mapped to browser actions that e.g., check for the existence of certain keywords or HTML elements.

The second part of the mapping is considered to be a static mapping since it is bound to a specific framework. It translates each of the WAAL constructs into executable API calls of the underlying framework so that these abstract actions can be executed in a real browser. More details about the WAAL language can be found in [9, 10].

VI. Experimentation on WebGoat

The techniques presented above have been applied to the WebGoat [11] platform. It consists of several deliberately insecure applications, called lessons, to illustrate each vulnerability type. We have applied our techniques to the "Stored XSS" lesson where we have access to a human resource management system with a database of profiles and classic actions, e.g., view, edit, delete. A stored Cross-site Scripting attack consists of a vulnerability that allows an attacker to store malicious (Java-)Script in the application so that it gets executed when a victim visits the corresponding web page.

For the reverse-engineering part, SIMPA detects all inputs and outputs of the lesson and builds a model in less than a minute. The only values provided by the user are the credentials. Given a security goal corresponding to the XSS attack, the model checker quickly finds a trace showing that the administrator can see each field of any profile after a modification by the owner, which is consistent with the application, but also a potential XSS vulnerability. Using WAAL, we were able to execute this trace on the application with an XSS payload as the parameter value and confirm that it is a real attack.

Operating on an annotated secure model for the above described WebGoat lesson, SPaCiTE applied 8 different semantic mutation operators (covering SQL injection, XSS attacks, and access control violations) and generated 54 mutated models. Executing the reported attack traces, SPaCiTE successfully executed 14 attacks (6 related to access control and 8 to XSS).

VII. Related Work

Each analysis component of the SPaCIoS project can be related individually with various state-of-the-art approaches.

The black-box model inference engine extends previous work to learn a parameterized model [5] by improving the algorithm for the security testing domain. Existing work [12, 13] usually considers the specification as available, but most of the time, this is not the case. Crawling modern web applications has been explored in [14] and [15] using DOM events but not for the purpose of test driver generation. Recently, crawling and vulnerability detection have been combined in [16].

Extracting a formal model from source code for use with a model checker has been also addressed in [17]. In contrast, our white-box modeling component produces models for verifiers specially built to check security properties.

Formal models and model checkers have long been used for test case generation. As surveyed in [18], generating test cases is usually based on satisfying structural criteria on the model (e.g., state or transition coverage). In contrast, SPaCiTE relies on a domain-specific fault model due to the lack of a strong relationship between such coverage criteria and fault detection effectiveness [19]. The idea of using mutation testing for test case generation was successfully applied for specification-based testing from AutoFocus or HLPSL models [20, 21]. Our work differs in that semantic mutation operators capture real vulnerabilities in web applications. Moreover, we do not stop after test generation but also provide a semi-automatic way to execute the generated test cases on real implementations.

Armando et al. [22] describe work closely related to SPaCiTE but for protocols instead of web applications. They start from an already insecure model described at HTTP level and map from abstract to concrete HTTP elements for executing test cases. The fully automatic procedure is achieved at the price of describing the model at the HTTP level.

VIII. Conclusion

We have presented two possible workflows of the SPaCIoS tool: inferring a model in black-box mode to automatically detect security flaws but also generating test cases from secure models. As we use a model-based approach, we have developed two methods of reverse-engineering web applications. Using these two complementary approaches, e.g., black-box with HTTP interactions and white-box from the source code, we have broadened the scope of the SPaCIoS tool to more applications. To validate the consistency of the model for security testing, we have briefly described the application of the tool as a proof-of-concept on WebGoat, where the reverse-engineered model has been model-checked. The attack found was executed and confirmed on the system using the execution engine. For cases where secure models were used, the mutation-based approach was successfully applied to such annotated models in order to generate and execute test cases that confirm vulnerabilities in the described WebGoat lesson.

References

[1] "SPaCIoS: Secure Provision and Consumption in the Internet of Services," http://www.spacios.eu/.

[2] "The AVANTSSAR Project: Automated Validation of Trust and Security of Service-Oriented Architectures," http://www.avantssar.eu.

[3] D. von Oheimb and S. Mödersheim, "ASLan++ – a formal security specification language for distributed systems," in *9th International Symposium on Formal Methods for Components and Objects (FMCO)*, ser. LNCS, vol. 6957. Springer, 2010, pp. 1–22.

[4] K. Hossen, R. Groz, C. Oriat, and J.-L. Richier, "Automatic generation of test drivers for model inference of web applications," in *Fourth International Workshop on Security Testing (SECTEST)*, 2013.

[5] K. Li, R. Groz, and M. Shahbaz, "Integration testing of distributed components based on learning parameterized I/O models," in *26th Internat. Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, ser. LNCS, vol. 4229, 2006, pp. 436–450.

[6] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[7] Y. Wang and I. H. Witten, "Induction of model trees for predicting continuous classes," in *Poster papers of the 9th ECML*, 1997.

[8] P. F. Mihancea and M. Minea, "jModex : Model extraction for verifying security properties of web applications," in *Proceedings of the 1st IEEE CSMR-WCRE Software Evolution Week*, 2014.

[9] M. Büchler, J. Oudinet, and A. Pretschner, "Semi-automatic security testing of web applications from a secure model," in *Sixth IEEE Int. Conf. on Software Security and Reliability (SERE)*, 2012, pp. 253–262.

[10] SPaCIoS, "Deliverable 3.4.1: Bridge components for different levels of abstraction," http://www.spacios.eu/, 2013.

[11] OWASP, "WebGoat Project," http://code.google.com/p/webgoat/, 2011.

[12] M. Zulkernine, M. F. Raihan, and M. G. Uddin, "Towards model-based automatic testing of attack scenarios," in *28th Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP)*, ser. LNCS, vol. 5775. Springer, 2009, pp. 229–242.

[13] Y. Hsu, G. Shu, and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," in *16th IEEE Internat. Conference on Network Protocols (ICNP)*, 2008, pp. 114–123.

[14] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri, A. Moosavi, G. von Bochmann, G.-V. Jourdan, and I. V. Onut, "Crawling rich internet applications: the state of the art," in *22nd CASCON*, 2012, pp. 146–160.

[15] G. Wu and F. Liu, "Web crawler for event-driven crawling of AJAX-based web applications," in *2nd International Conference on Emerging Technologies for Information Systems, Computing, and Management (ICM)*, ser. LNEE. Springer, 2013, vol. 236, pp. 191–200.

[16] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: a state-aware black-box web vulnerability scanner," in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 26–26.

[17] 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.

[18] G. Fraser, F. Wotawa, and P. Ammann, "Testing with model checkers: a survey," *Softw. Test. Verif. Reliab.*, vol. 19, no. 3, pp. 215–261, 2009.

[19] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *16th Int. Conf. on WWW*, 2007, pp. 667–676.

[20] F. Dadeau, P.-C. Héam, and R. Kheddam, "Mutation-based test generation from security protocols in HLPSL," in *4th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST)*, 2011, pp. 240–248.

[21] G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations," in *4th Int. Conf. Formal Engineering Methods (ICFEM)*, ser. LNCS, vol. 2495, 2002, pp. 471–482.

[22] A. Armando, G. Pellegrino, R. Carbone, A. Merlo, and D. Balzarotti, "From model-checking to automated testing of security protocols: Bridging the gap," in *6th International Conference on Tests and Proofs (TAP)*, ser. LNCS, vol. 7305. Springer, 2012, pp. 3–18.