Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2012,3
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Semi-Automatic Security Testing of Web Applications from a Secure Model

Matthias Büchler, Johan Oudinet, Alexander Pretschner

2012

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

Please note:
This Report has been published on the Internet under the following Creative Commons License:
http://creativecommons.org/licenses/by-nc-nd/3.0/de.

Semi-Automatic Security Testing of Web Applications from a Secure Model

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 are a major target of attackers. The increasing complexity of such applications and the subtlety of today's attacks make it very hard for developers to manually secure their web applications. Penetration testing is considered an art; the success of a penetration tester in detecting vulnerabilities mainly depends on his skills. Recently, model-checkers dedicated to security analysis have proved their ability to identify complex attacks on web-based security protocols. However, bridging the gap between an abstract attack trace output by a model-checker and a penetration test on the real web application is still an open issue. We present here a methodology for semi-automatic testing web applications starting from a secure model. First, we mutate the model to introduce specific vulnerabilities present in web applications. Then, a model-checker outputs attack traces that exploit those vulnerabilities. Next, the attack traces are translated into concrete test cases by using a 2-step mapping. Finally, the tests are executed on the real system using an automatic procedure that may request the help of a test expert from time to time. A prototype has been implemented and evaluated on WebGoat, an insecure web application maintained by OWASP. It successfully reproduced RBAC and XSS attacks.**

I. Introduction

Since web applications handle sensitive data, ensuring their security is important and a prerequisite for their use in business contexts. Web applications usually are not monolithic but consist of several distributed components. During the development of the use communication protocols and the web components, different tools and programming languages may be used. White-box penetration testing tools like (Kieyzun et al., 2009) usually require that all applications are developed in the same language (e.g., PHP for Ardilla) which is usually not the case in distributed environments (not mentioning that having access to the source code of every component is also an issue in practice). Unfortunately, current black-box penetration testing tools are not really effective due to the weaknesses of the crawling step that misses lots of potential interaction with the user (e.g., the output page may depend on the parameters provided by the user and it is hard to guess those parameters in general); see (Doupé et al., 2010) for an evaluation of such penetration scanners that show evidence of those weaknesses.

In our work, we assume there is a formal model $\mathcal{M}$ for the specification of the System Under Validation (SUV) (coming up with such model is a classical issue in Model-Based Testing (MBT) and more details are given in subsection IV-C). This model is secure as it does not violate any of the specified security goals (i.e., a model-checker will report $\mathcal{M} \models \varphi$ for all security properties φ defining the security goals of the model). Otherwise, it would mean the specification of the SUV is known to be insecure and there would be no reason to test security properties of any implementation that refines $\mathcal{M}$. Note that a modeler can make use of existing tools (Armando and Compagna, 2008; Turuani, 2006) to fix an insecure model, which should be done before starting the implementation.

Taking advantage of the model describing the specification could help black-box penetration testing tools in discovering vulnerabilities issues. Recently, model-checkers dedicated to security analysis have proved their ability to identify complex attacks on web-based security protocols (Armando et al., 2011). However, bridging the gap between an abstract attack trace output by a model-checker and a penetration test on the real web application is still an open issue. We present here a methodology for semi-automatic testing web applications starting from a secure model. First, we mutate the model to introduce specific vulnerabilities present in web applications. Then, a model-checker outputs attack traces that exploit those vulnerabilities. Next, the attack traces are translated into concrete test cases by using a 2-step mapping. Finally, the tests are executed on the real system using an automatic procedure that may request the help of a test expert from time to time. To evaluate our approach, we implemented a prototype[1] for a RBAC and an XSS lesson in WebGoat[2], an insecure web application maintained by OWASP.

Problem Statement: We address several problems related to a complete method for testing web applications from secure models, which describe the system at an abstract level. First, how to make use of a model-checker to generate interesting test cases from a secure model. Second, how to semi-automatically instantiate such interesting test cases from the level of the model to the level of the implementation. Third, how to guide a penetration tester when the Test Execution Engine (TEE) needs his help to execute one step of the test cases.

Contributions: The main contribution using the methodology presented in this paper is the ability to exploit a model describing a Web application at the browser level (i.e., without low-level communication knowledge) to guide a penetration tester in finding attacks based on logical vulnerabilities (e.g.,

[1] A demo video is available at: http://zvi.ipd.kit.edu/26_500.php
[2] https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

a missing check in a Role-Based Access Control (RBAC) system, non-sanitized data leading to Cross-Site Scripting (XSS) attacks). The main parts of the approach are as follows:

- *Mutation operators related to real vulnerabilities in Web applications*: Starting from a secure model, we provide a few mutation operators that reflect the potential presence of specific vulnerabilities and allow a model-checker to generate attack traces that exploit those vulnerabilities.
- *Web Application Abstract Language (WAAL)*: The instantiation of abstract attack traces into executable test cases is done by using a 2-step mapping approach. First, the message-based attack trace is mapped to an intermediate language for Web applications, called WAAL. As this intermediate language is independent from the web application under test, test experts do not need to provide the second mapping, from WAAL to source code.
- *Semi-automatic Test Execution Engine (TEE)*: The TEE automatically executes test cases at the browser level, using the Selenium framework[3]. If an action cannot be performed at the browser level (e.g., an item cannot be selected from a drop-down menu), the TEE may ask for the help of a test expert to provide corresponding HTTP requests; hence the semi-automatic procedure.

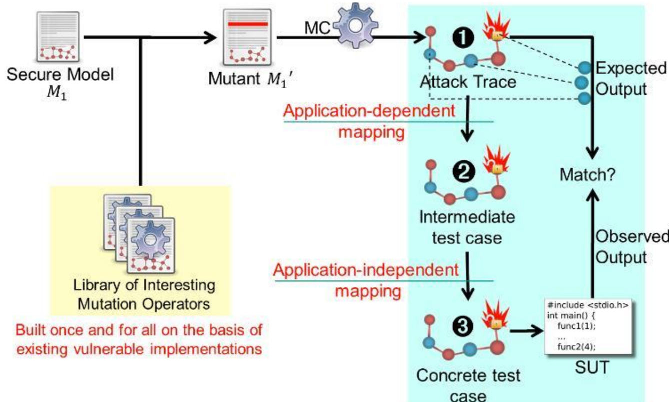

Figure 1. Overview of the testing process

Organization: The paper is organized as follows. In section II, we present how to generate abstract test cases from a secure model (i.e., the left part of Figure 1). In section III, we explain the instantiation and execution of the generated attack traces (i.e., the right part of Figure 1). This testing process has been applied to some lessons in WebGoat and we show the experimental results in section IV. Finally, we review the related work in section V and conclude in section VI. A description of the tool architecture is presented in the Appendix.

II. Automatic Generation of Test Cases From Secure Model

In this section, we describe the general approach to automatically generate test cases from a secure model. The principle is described in subsection II-A. Then, the method is illustrated in subsection II-B on a concrete web application: a lesson in WebGoat that manages user profiles.

A. Principle

We assume a secure model $\mathcal{M}$ as a starting point to the test case generation (i.e., $\mathcal{M} \models \varphi$ with $\varphi \in \Phi$ for a set of interesting security properties Φ). The security properties Φ (e.g., confidentiality, authenticity, authorization) are assumed to be provided together with the secure model. In our tools, they are formalized using the AVANTSSAR Specification Language (ASLan++) as it is the input language for model-checkers dedicated to security analysis (AVANTSSAR, 2011).

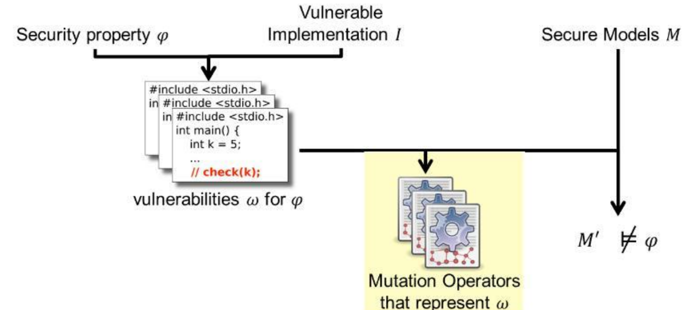

Figure 2. Process to build a library of interesting mutation operators

The model itself is built using abstract messages that are defined by the modeler. These messages represent common actions a user of the web application can perform. The idea is that these abstract messages are sent to the server to tell it which action the client wants to perform. For example, in the WebGoat lessons that we consider later, these abstract messages may represent the following actions: log in to the web application, view profiles of different users, delete profiles, update profiles, and so on. Thus, the modeler does not care about details at the browser/protocol level but only about abstract messages that represent web application actions.

As the model is considered secure, all its traces satisfy all specified security properties $\varphi \in \Phi$. Thus, the model-checker does not exhibit any counter-example. Checking $\mathcal{M} \models \neg\varphi$ is of limited help either since the generated counterexamples constitute *all* traces of the model—a filter for test case selection is missing. Hence, in order to use a model-checker for testing purposes, we must modify either $\mathcal{M}$ or φ such that the model-checker returns counter-examples, which then can be turned into test cases. If we modify φ into φ' such that $\mathcal{M} \not\models \varphi'$, the model-checker will return a trace from $\mathcal{M}$ that is irrelevant to check the violation of φ (because $\mathcal{M} \models \varphi$). As a consequence, we modify $\mathcal{M}$ into $\mathcal{M}'$ by using a predefined set of mutation operators that are relevant for the SUV. Therefore, if $\mathcal{M}' \not\models \varphi$, the model-checker will return counter-examples that are interesting for testing if the implementation has such security holes.

Figure 2 shows how the library of interesting mutation operators is created from a manual analysis of a learning set of vulnerable web applications. First, each vulnerability ω is associated to the security properties φ that are violated by attacks exploiting this vulnerability. Then, we try to inject this

[3]http://seleniumhq.org/

vulnerability into a secure model of the application such that a model-checker can exhibit an attack that exploits this vulnerability. If this is possible, the corresponding mutation operator is considered as interesting for "injecting" the vulnerability ω into a secure model. As every vulnerability is associated to security properties, it is possible to select a set of interesting mutation operators for a specific security property.

After having applied a mutation operator to an original model, the model-checker may provide a trace from this mutated model that violates a security property. This trace is called an *attack trace* because it shows which sequence of abstract messages have to be exchanged in order to lead the system to a state where the security property is violated. To illustrate this process, we consider a concrete example based on WebGoat.

B. Test case generation for authorization flaws in WebGoat

Access control aims at ensuring that confidential data are restricted to authorized users only. If the access control is enforced by the presentation layer only (i.e., hiding buttons or links that are not authorized to the user), then an intruder can get access to the confidential data by using a Direct Request attack[4] (i.e., asking directly the resource via a GET or POST request). An example of this vulnerability can be found in the WebGoat "Bypass Data Layer" lesson, where a malicious employee can access other employees' profiles even though he is not authorized to see them.

In this WebGoat lesson, a user must log in before viewing an HTML page that contains a list of employee's names together with HTML buttons, which represent actions that can be performed: search staff, view profile, delete profile, and log out. Only names of employee profiles the user is authorized to view are displayed.

A secure model of this application has been developed in ASLan++. Even though ASLan++ was created for security protocols, the language can naturally be used to model web applications as well. For a full description of ASLan++, please refer to (AVANTSSAR, 2011). For example, the actions a user has to perform to access a profile are modeled as follows:

```
body { % of User
  % Login
  Actor ->* S: login(Actor,password(Actor,S));
  S -> Actor: listStaffOf(Actor);
  % ViewProfile
  if (Actor->isAuthorizedToView(?A)) {
    Actor *-> S: viewProfileOf(A);
    S *->* Actor: A.?Profile;
}
```

A user (`Actor`) sends her credential to a web server (`S`), via a confidential channel (`->*`), and gets back, only if her credential is accepted by `S`, the page `listStaffOf(Actor)` that lists the user names whose profiles she is authorized to view. Then, she asks for viewing `A`'s profile, via an authentic channel (`*->`) as she is already logged in. If she is authorized to view `A`'s profile (as defined by the RBAC system), `S` will send her the profile over an authentic and confidential channel (`*->*`).

[4] http://cwe.mitre.org/data/definitions/425.html

The communication represents the important steps a user has to perform to view a profile. We call such model a message-based model because it models the different abstract messages (e.g., `login`, `viewProfileOf`) that are exchanged when a user performs an action at the frontend of the web application.

The server entity is modeled as well such that the defined security goals are not violated. For example, the `viewProfile` request is handled as follows:

```
% ViewProfile action
on(?U *-> Actor: viewProfileOf(?A)
   & authenticated(?U)
   & ?U->isAuthorizedToView(?A)): {
     % Get A's profile
     select{on(hasProfile(A,?Profile)):{}}
     Actor *->* U: A.Profile;}
```

The environment is also part of the model. Here, we show a subset of the real system environment (only two authorized users, `tom` and `jerry`, are represented).

```
body { % of Environment
  % Set profiles
  jerry->hasProfile(jerryProfile);
  tom->hasProfile(tomProfile);
  % Jerry is authorized to visit tom's profile
  jerry->isAuthorizedToVisit(tom);
  % Let assume that tom has been compromised
  % Then i can play tom's role
  dishonest(tom);
  iknows(inv(ak(tom))); iknows(inv(ck(tom)));
  iknows(inv(pk(tom)));
  i->isAuthorizedToVisit(tom);
  iknows(password(tom, server));
  iknows(tomProfile);
  % Create a session between the honest user
  % jerry and the web server
  new Session(jerry, server);}
```

In the WebGoat lesson description, a penetration tester must log in as `tom` and try to access to an unauthorized profile for this user, for example `jerry`'s profile. By default, there is only one dishonest agent in ASLan++, called `i`. Marking `tom` as a compromised user allows the model-checker to consider the scenario when a penetration tester can log in as `tom`. Thus, the environment described in the model should reflect as much as possible the real environment.

Finally, the goal that a profile can only be accessed by an authorized user is defined as follows:

```
goals
  secret_profiles:
  forall A P. [](iknows(P) & A->hasProfile(P)
                 => i->isAuthorizedToView(A));
```

which means if a malicious user `i` gets access to the profile `P` (i.e., `iknows(P)`) that belongs to user `A` (i.e., `A->hasProfile(P)`), `i` must be authorized to do so (i.e., `i->isAuthorizedToView(A)`).

Now that we have the security goals and a secure model, we can apply the test case generation method presented before. The first step consists in mutating the secure model to inject potential vulnerabilities that are relevant for the defined security goals. Here, we have a secrecy goal that relies on a RBAC

system. A missing check is a common authorization flaw that could be exploited here. Thus, the tool uses the corresponding mutation operator: removing an authorization check at the server entity. After applying this mutation operator, the modified part of the model looks like follows:

```
% ViewProfile action
on(?U *-> Actor: viewProfileOf(?A)
   & authenticated(?U)
%% Authorization check has been removed:
%%%%%    & ?U->isAuthorizedToView(?A)
  ): {
    % Get A's profile
    select{on(hasProfile(A,?Profile)):{}}
    Actor *->* U: A.Profile;}
```

Giving this mutated model to the Cl-Atse model checker (Turuani, 2006) an attack is found and reported as follows:

```
VIOLATED:
  secret_profiles[A=jerry , P=jerryProfile]  % on line 122

MESSAGES:
  <tom>  ->*  server: login(tom,password(tom,server))
  server ->  <tom> : listStaffOf(tom)
  <tom>  *->  server: viewProfileOf(jerry)
  server *->* <tom> : jerry.jerryProfile
```

The model checker reports a violation of the `secret_profiles` property. In addition the `MESSAGES` section shows a sequence of messages that drives the model to a state where the reported security property is violated.

In the first step a user, pretending to be `tom`, logs into the server by sending the `login` message which contains the shared password. As the password is correct, the server agent (`server`) acknowledges the request by sending back the staff list of `tom`. Then, the user `tom` tries to request the specific profile of `jerry` that he should not be allowed to view according to the RBAC system (the model specifies that only `jerry` is authorized to view his profile). Actually, `tom` tries this request because he is modeled as a compromised user and thus an intruder acts maliciously on behalf of `tom`. As the authorization check `isAuthorizedToView` at the server side has been removed by the mutation operator, the request by `tom` is accepted and `jerry`'s profile is sent back to `tom`.

The abstract trace must be instantiated and executed to find out if the real implementation is vulnerable to this attack. This is the subject of the next section.

III. Instantiation and Execution of Test Cases

The mapping of the abstract attack trace to executable source code is a multi-step process because it consists of application-dependent and application-independent information. To separate the two kinds of information, we add an additional intermediate level (❷ in Figure 1) in between the abstract attack trace layer (❶ in Figure 1) and the implementation layer (❸ in Figure 1). These three layers have different purposes. Layer ❶ describes the abstract attack trace as it is given by the output of the model checker. The abstract attack trace consists of a sequence of messages that are exchanged between the defined agents. Layer ❷ describes an intermediate layer where the same abstract attack trace of layer ❶ is described using actions of WAAL, a dedicated language for web applications. WAAL is a language to describe how exchanged messages between agents can be generated and verified in terms of actions a user performs in a web browser. Finally layer ❸ describes the instantiated attack trace in terms of source code. In addition it shows how the TEE reacts if an error or exception occurs during the execution of the attack trace.

First, the TEE is described in subsection III-A to have a clear understanding how the abstract test cases must be instantiated to be executable. Then, after describing WAAL in subsection III-B, we present in subsection III-C the first mapping from application-dependent messages to the intermediate level. Finally the second mapping, from the intermediate level to executable test cases, is presented in subsection III-D. As WAAL actions are application-independent, the last mapping can be reused for testing other web applications. Both mappings are illustrated with an application to the WebGoat lesson presented in subsection II-B.

A. Test Execution Engine (TEE)

The TEE is responsible for running test cases and reporting verdicts. A *test case* is a sequence of descriptions of controlled and observed messages. Controlled messages are also called stimuli and observed messages are also called reactions. As such, with the reactions, a test case encodes the expected behavior with respect to the stimuli.

Running a test means *applying the stimuli* to the SUV and *observing* the SUV *reactions* (the actual reactions). Building a *verdict* means to compare the actual reactions to the expected reactions. If they conform, we say the test passes. If they do not, we say that the test fails. The result of this comparison is called the verdict. In our context, we generate *attack traces*. If we successfully reproduce an attack on a SUV, then our terminology applies as follows. As the expected reaction says that the attack should not be reproduced by the SUV (which is in conformance with the SUV's specification), then we say that the attack has been reproduced, but the test has failed.

Having in mind this terminology, let us describe now the language used at the intermediate level.

B. Web Application Abstract Language (WAAL)

WAAL is an abstract language for web application actions at browser level. The purpose of this language is to define actions that an end user can perform from a browser to either send messages to a Web server or check its responses. Thus, WAAL actions are split into two sets: Browser Interface Actions (BIAs) and Verification Actions (VAs).

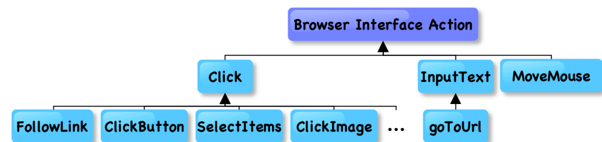

Figure 3. Browser Interface Actions (BIAs)

Browser Interface Actions (listed in Figure 3) represent a small but complete set of atomic actions that a user can perform when he uses a web application (e.g., follow a link, click on a button, type text into a text field). More complex actions can be described by a combination of such atomic actions. For example, log in via a form may correspond to the sequence: select the name from a menu, type the password into a text field, and click on the login button. Since it works at the Browser level, BIAs are close to API methods from Selenium, a Web application testing framework. However, BIAs are not API methods at source code level but abstract browser actions and therefore they are technology independent.

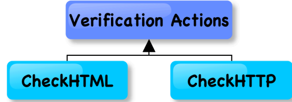

Figure 4. Verification Actions (VAs)

Verification Actions (listed in Figure 4) are used to verify whether an observed response matches with an expected one. A user can either verify the received message at HTML or HTTP level. The verification is performed according to a user provided criterion.

The BIA and VA sets define the foundations for WAAL:

$$WAAL = (BIA^* \times VA^*)^*$$

In other words, a valid word in WAAL is a sequence of actions that either produces (BIA^*) or verifies (VA^*) protocol-level messages. Note that we consider sequences only (and not trees) because this language is intended to represent the abstract attack trace at the browser level. Since a trace from a model-checker is an abstract message sequence, a sequence of actions at browser level is sufficient to represent such traces.

C. Mapping from abstract model level to browser level

1) General principle: The output of the model checker is an abstract attack trace that consists of a sequence of exchanged messages. Each message m has a sender agent $\mathcal{S}$, a receiver agent $\mathcal{R}$ and a channel $\mathcal{C}$. Thus, the input layer $\mathcal{L}_1$ for the mapping to WAAL is defined as follows:

$$\mathcal{L}_1 = (A \times C \times A \times M)^*$$

where A is the set of agents, C is the type of channel used (confidential, authentic, or both), and M is the set of abstract messages exchanged between two agents.

The mapping τ_1 maps each message m (together with its sender, receiver and channel) to a pair of sequences that generates and verifies m:

$$\tau_1 : (A \times C \times A \times M) \rightarrow (BIA^* \times VA^*)$$

The actual mapping depends on the sender and the receiver. Each agent described in the model is either part of the SUV — the TEE can observe his behavior — or is simulated (stubbed) by the TEE. The former kind of agent is denoted by the set A_o, for observed agents, while the latter is denoted by the set A_s, for simulated agents. Partitioning the agent set A into A_o and A_s is the responsibility of the test expert.

Given these two sets, the sequence of BIAs for $\mathcal{S} \rightarrow \mathcal{R} : m$ is constructed as follows:

$$\begin{cases} (bia_1, bia_2, \ldots, bia_n), & \text{if } \mathcal{S} \in A_s \\ (), & \text{if } \mathcal{S} \in A_o \end{cases}$$

where $n \in \mathbb{N}$ and $bia_i \in BIA$ for all $1 \leq i \leq n$. Thus, if the sender $\mathcal{S}$ is a simulated agent, the message m is mapped to a sequence of BIAs such that the message m is generated by a web browser after executing this sequence. If the sender is an observed agent, the TEE does not need to generate anything.

In addition to A_o and A_s, the sequence of VAs also depends on an assumption about the channel, namely whether sent messages can be assumed to be delivered unmodified. This assumption is called *integrity assumption*.

$$\begin{cases} (), & \text{if } \mathcal{S} \in A_s \wedge \text{integrity} \\ (va_1, va_2, \ldots, va_n), & \text{otherwise} \end{cases}$$

where $n \in \mathbb{N}$ and $va_i \in VA$ for all $1 \leq i \leq n$. Thus, a message m is mapped to a sequence of VAs such that a browser can verify the received message m by executing this sequence. The only case where the TEE does not need to verify m is when m has been sent over an integrity channel by a simulated agent.

In addition to mapping every message from the attack trace to sequences of actions in WAAL, the test expert must also provide an initialization block (*BIA_0* in Figure 5) in order to prepare the execution of the attack trace. This initialization block is also described as actions in WAAL.

Let us now give a concrete example of this mapping, by using again the WebGoat lesson on authorization flaws.

2) Application to WebGoat: According to the model, the agent `server` is in A_o as it is part of the SUV and the agent `tom` is in A_s as it is a compromised user and therefore must be controlled by the TEE. For our example, we also assume the integrity of messages sent over the channels. Thus, messages generated by simulated agents do not have to be verified.

The abstract attack trace found by the model-checker (presented in subsection II-B) consists of four messages (`login`, `listStaffOf`, `viewProfileOf`, `profileOf`). Listing 1 shows the mapping of these four messages to sequences of actions in WAAL; Only the BIAs and VAs relevant to the attack trace are shown.

In addition to the mapping of the attack trace, the initialization block in WAAL (*BIA_0*) provides a way to put the system into a state suitable to run the attack trace. For the WebGoat example, this initialization block follows some links to reach the login page of the lesson under test.

D. Mapping from WAAL to executable source code

Once the attack trace is translated into WAAL actions, the remaining step to be able to execute the test case is to map these WAAL actions into executable statements. In contrast to the first mapping τ_1 (from abstract messages to WAAL actions) that is application dependent, the second mapping τ_2 (from

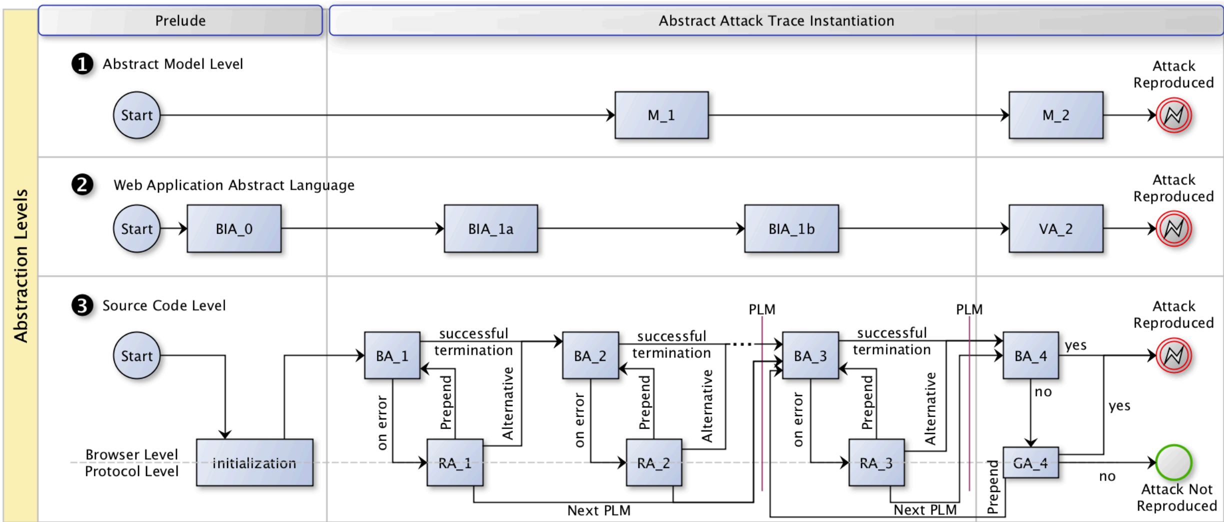

Figure 5. Instantiation and Execution Methodology

```
τ1(login(usr, pwd)) =
  ((selectItem(employeeList, usr),
    intputText(passwordField, pwd),
    clickButton(login)), ())

τ1(listStaffOf(usr)) = ((),
  (checkHTML(criterionFor(listStaffOf(usr)))))

τ1(viewProfileOf(usr)) =
  ((selectItem(profileList, usr),
    clickButton(ViewProfile)), ())

τ1(profileOf(usr)) = ((),
  (checkHTML(criterionFor(profileOf(usr)))))
```

Listing 1. Mapping of WebGoat abstract actions to WAAL

WAAL to source code) is done once and for all, except if the technologies used by the TEE change.

1) General principle: At the source code layer, two API interfaces are used in cooperation, even though they operate on different abstraction levels. The first API works at the browser level and is then close to WAAL, which makes the translation of WAAL actions to this API easier. The second API works directly at the protocol level and is then close to Web application communication protocol. The second API is needed only if an action cannot be performed by the first API. In that case, the TEE may request the help of a test expert for providing the corresponding protocol-level message.

In Figure 5, there are three kinds of blocks at the source code level: Browser Action (BA), Guide Action (GA), and Recovery Action (RA). A BA block corresponds to an action performed on a browser. A RA block corresponds to a recovery action performed after a failure from a BA, related to a Browser Interface Action. A failure in a BA block is either a runtime exception (e.g. a browser object where an action should be invoked does not exist) or the response of the BA block corresponds to a runtime exception from the SUV (e.g. the webserver returns an "authentication required" response instead of the desired webpage). A GA block occurs when a BA block related to a Verification Action fails. GA and RA blocks belong to either the browser or the protocol level, depending on the failure that triggers those actions. Both GA and RA may ask a test expert to provide additional information.

The mapping $\tau_2 : BIA \cup VA \rightarrow (BA \times RA)^* \cup (BA \times GA)^*$ maps each WAAL action to a sequence of BA, RA and GA with $\tau_2(a) \in (BA \times RA)^*$ if $a \in BIA$ and $\tau_2(a) \in (BA \times GA)^*$ if $a \in VA$.

If the TEE can successfully execute every BA block, which is done in a fully automatic way, then the verdict is determined as follows: if the actual reactions of the SUV conform to the expected reactions of the test cases — this verification is done by the BA blocks related to VA actions —, the attack has been reproduced and therefore the test has failed; otherwise, the test has passed.

However, a BA block may fail due to several reasons of different nature. For example, an input element is disabled, in read only mode, or its `maxLength` attribute is set to a value smaller than the size of the text to type in. Another example is a button that is disabled or totally missing, and therefore the BA block cannot click on it. For some failures of this kind, it might be possible to execute a recovering action and continue the test execution.

If an error occurs when executing a BA block, the TEE changes its operational mode and a RA (resp. GA) block is executed in order to recover from this error (resp. review the decision). There are three different ways of recovering after a BA block has failed: (i) prepend missing information to the BA block and execute it again; (ii) find an alternative way to execute the BA block and resume just after it; (iii) move to the protocol level, provide the corresponding message,

and resume after the next protocol-level message (which may be after several BA blocks). For the following examples of these recovering methods, the browser level represents HTML elements including their actions, and the protocol level is HTTP.

As an example for the *prepend* case, let BA be an action to check the content of a webpage. This action may fail because the user is not authenticated and first has to provide credentials. In the case of basic access authentication, this request for credentials can be automatically detected and therefore it is not necessary to provide this step as WAAL actions. Thus, a possible action in GA could be that the TEE asks the test expert for the credentials or that they are read from a configuration file. Then, the TEE reconfigures the used component by adding the credentials and re-requests the website again. Requesting a website and adding credentials can both be performed at the browser level.

For the *alternative* case, let us consider an HTML button element that triggers an event if the user clicks on it and this event is the execution of a defined JavaScript function. If the HTML button is disabled, the click event can not be triggered by the BA block. A possible alternative action, performed by the corresponding RA block, is to execute the JavaScript function directly, by using a different API call.

An example where the TEE has to switch to the protocol (HTTP) level is the following one. Assume that a BA block tries to select an element from a list and sends this value to the server by clicking on a button. This action may fail because the element is not present in the list. In that case, the TEE presents some sample HTTP messages to the test expert (e.g., by generating the HTTP messages corresponding to choosing another element from the list). Then, the TEE asks the test expert to provide the correct HTTP message. This message is sent and the BA block that follows this HTTP message is executed afterwards. It is worth noting that the underlying assumption when a RA creates some HTTP samples is that the agent state may be restored afterwards. Thus, as soon as the TEE intercepts and drops the HTTP requests, the RA block can generate as many samples as possible.

2) Application to WebGoat: For our example, we consider the Selenium Framework at the browser level and the Apache HTTPComponents project[5] at the protocol level. Selenium provides WebElement objects to represent HTML elements and WebDriver objects to represent "the Browser" at the source code level. These objects provide API functions to find an element in the browser object (`HtmlUnitDriver.findElement()`), to access the current webpage of the browser (`HtmlUnitDriver.getPageSource()`), or to perform a click action on an HTML element (`WebElement.click()`).

Due to space constraints we only provide one example mapping for WebGoat. Listing 2 shows how the WAAL action `selectItem`, that is part of Listing 1, is mapped to Selenium source code. The result is a Java function that takes as parameters: a connection object, the name of the list and the name of the item that should be selected from that list.

[5]http://hc.apache.org/

```
Connect selectItem(Connect conn,String listID ,String item){
  try {
    conn.convertToHtmlDriver();
    // Get the corresponding list
    c1 = "//a[contains(text(),'" + listID + "')]";
    list = conn.driverHtml.findElement(By.xpath(c1));
    // Select the item from the list
    c2 = "//a[contains(text(),'" + item + "')]";
    list.findElement(By.xpath(c2)).click();
  } catch (Exception e) {
    // Activate RA Block
  }
  return conn;
}
```

Listing 2. Mapping of τ_2 $(selectItem(listID, item))$

The function then executes the following actions: (i) find the HTML list element, (ii) click on the corresponding item from this list. The try-catch block captures runtime exceptions and executes the corresponding RA block.

IV. Evaluation: Application to WebGoat

In the evaluation, we investigated the following three research questions:

- RQ1. Can we successfully exploit a vulnerability at the concrete level?
- RQ2. How many times do test experts have to be guided?
- RQ3. What is the advantage of modeling the system at the abstract message level instead of the protocol level?

A. Methodology

We implemented a prototype of our methodology to answer these questions. This prototype is based on ASLan++ for the modeling language, CL-AtSe for the model-checker, Selenium and Apache HTTPComponents for the TEE, and the analysis of two WebGoat lessons from different domains (RBAC and XSS) for developing the mutation operator library. This library contains a mutation operator that represents a missing authentication check (the vulnerability present in the RBAC lesson) and a mutation operator that introduces a non sanitizing action according to the vulnerability in the XSS lesson.

Then, we applied this prototype on four different lessons (two about RBAC and two about XSS) of the WebGoat application, including the two lessons used for the mutation operator library. Since models of these four lessons did not exist, we manually built a secure model for each lesson according to our experience, the descriptions, and the source code of the lessons. We applied the mutation operator for missing authentication checks to both the lessons "Bypass Business Layer Access Control" (Lesson 1) and "Bypass Data Layer Access Control" (Lesson 2). The mutation operator for non sanitizing actions was applied to the lessons "Stored XSS" (Lesson 3) and "Reflected XSS" (Lesson 4). Next, the model checker produces abstract attack traces for all four lessons. Using the 2-step mapping described in section III, we instantiated these attack traces using the Selenium Framework and the Apache HTTPComponents. Finally, we executed the instantiated attack traces to check whether they are reproducible on the SUV.

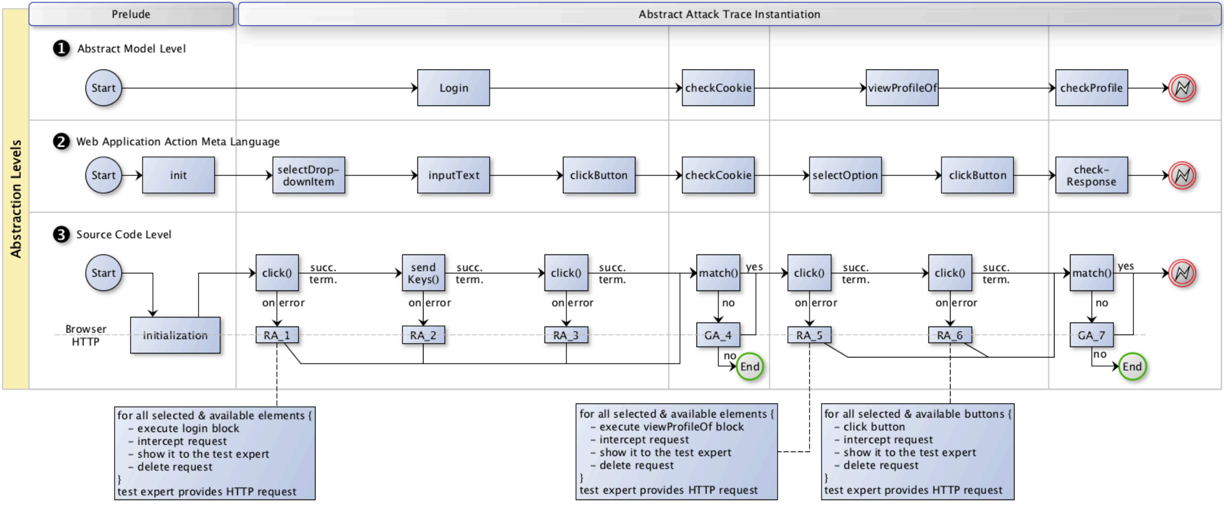

Figure 6. Instantiation

B. Results

A short answer to the first question is yes, we were able to successfully exploit the vulnerabilities present in the four WebGoat lessons we analyzed. We executed the instantiated attack traces produced as described in subsection IV-A. The attack traces are successfully executed because the WebGoat lessons contain the vulnerabilities captured by the applied mutation operators. Successfully means: `tom` can delete his profile even though he is not authorized to do so (Lesson 1), `tom` gets access to `jerry`'s profile even though he is not authorized to view this profile (Lesson 2), `tom` can store javascript code in his profile that is successfully executed when `jerry` views `tom`'s profile (Lesson 3), and `tom` can enter javascript code as a search query expression that is returned as part of the response and finally executed (Lesson 4).

Our approach can successfully be applied to all these lessons because the existing vulnerabilities in such lessons can be expressed and described as mutation operators. Nevertheless, there are other lessons (e.g., HTTP splitting attacks) that require details on modeling communication at HTTP level. Since our approach is based on models using high level browser actions, such attacks are not addressed by our approach.

To answer the second question, we execute the instantiated attack traces and compare the number of messages that the test expert has to provide to the total number of generated messages. For Lesson 1, the test expert has to be guided only for the second request message that asks for `jerry`'s profile. The reason for this is that `jerry`'s name is not present in the web page and hence cannot be selected by the Selenium framework. Therefore, the test expert is guided with messages for other profiles and finally has to provide the correct request message. Another reason why only one message needs guidance from the test expert is that many messages belong to the preparation phase, which relies on functionality of the WebGoat lesson. The attack itself happens in the last request and is therefore the only message that potentially needs the help of the test expert. In our case, the difference between provided sample messages (to guide the test expert) and the desired messages (as specified in the attack trace) is rather small and only differ in one numerical identifier. Assuming that the mapping from textual representation of the profile to its numerical ID is given or can be learned (e.g., from another webpage of the lesson), the desired message may be generated automatically after some structural analysis. This is actually very specific to the considered WebGoat lesson and has not been implemented in our prototype yet. Nevertheless, our methodology guides the test expert to come up with the correct desired message.

For Lesson 2, exactly the same reasoning applies with the minor detail that not the employee ID but rather the action name has to be provided by the test expert. Despite that, there is no fundamental difference to Lesson 1.

For Lessons 3 and 4, in addition to the attack trace, a database of (javascript code, verification code) tuples is needed. The javascript code is injected in every possible input element of non-sanitizing actions. Then, the verification code is used to check if the corresponding javascript code has been executed successfully. At the browser level all necessary actions can be performed automatically. The attack traces for Lessons 3 and 4 are instantiated for each (javascript code, verification code) tuple. The TEE does not need to be guided for both lessons.

For assessing the benefit of modeling the system at the browser level instead of the protocol level, we compare the number of needed parameters at each layer of abstraction (see Table I), as they appear in our lessons. The column “model” shows how many parameter we need for the message at the modeling level. The same is done for actions at WAAL level, and for the HTTP protocol level. For example, there are only 2 parameters (username, password) at the model level for the `login` message. At WAAL level we need 5 parameters (`dropDownID`, `username`, `passwordID`, `password`, and `login` button ID) to express the same action, whereas the corresponding HTTP message consists of 13 header

lines and 1 content section. The small number of parameters at the model level with regards to the number of parameters at the protocol level is an indication for improved understandability at the model level.

Table I
NUMBER OF PARAMETERS PER ABSTRACTION LEVEL

Message	Abstraction level		
	model	WAAL	protocol
`login`	2	5	14
`listStaffOf`	1	1	7
`viewProfileOf`	1	3	14
`profileOf`	1	1	7
`deleteProfileOf`	1	3	14
`editProfileOf`	1	16	14
`searchProfileOf`	1	4	14

C. Discussion

The provenance of models is always an issue in MBT and it is unclear if MBT is cost-effective due to the necessity of managing two artifacts: code and model. Nevertheless, there also is solid evidence for the merits of this approach (Grieskamp et al., 2011). Indeed, the high-level abstraction used by the models in our approach makes it possible to build a secure model from a web application by just using a browser, without looking at the HTTP communication. In fact, we needed 1 PM to build the first model and only 2 weeks to build the three other models, which indicates that a secure model can be built in a rather small amount of time by someone familiar with the SUV. There are also existing results to infer a model directly from a web application (Halfond et al., 2009).

We analyzed the source code of WebGoat to identify some vulnerabilities to turn them into mutation operators for ASLan++ models. However, this source code analysis is not necessary if mutation operators for the targeted vulnerabilities are already available. Indeed, the mutation operators were successfully applied to the new models to find further attacks.

The number of parameters in Table I may be misleading. For example, for the `profileOf` message, there is only 1 parameter in both the model abstraction and WAAL, but it corresponds to the user name in the model and to a criterion for distinguishing the profile inside a HTML page, which is more complex than just the user name. In general, parameters at a lower level are more complex to define than parameters at a higher level of abstraction. Note also that the abstract `editProfileOf` message corresponds to changing any mutable field from the profile; hence the 16 WAAL parameters since there are 13 fields in a profile.

V. RELATED WORK

Our work is at the intersection between MBT and penetration testing (pentesting). As we rely on mutation operators to introduce implementation-level vulnerabilities into the secure model and on model-checkers to generate attack traces, we describe here works related to mutation testing and security testing from a model-checker.

On one hand, security testing is usually performed by penetration testers that either use manual techniques, based on their knowledge and by following guidelines like the OWASP testing guide[6], or automated techniques thanks to penetration testing tools[7]. Such tools differ from our work as they do not rely on models for generating test cases. Doupé et al. (2010) evaluated such "point-and-click pentesting" tools and found that the crawling part (discovering new pages) is a critical and challenging task for these tools that determines the overall ability to detect vulnerabilities by black-box web vulnerability scanners. One way to overcome this weakness is to use a white-box testing approach, dedicated to applications written in a specific language. For example, the Ardilla tool (Kieyzun et al., 2009) looks for SQL injections and XSS attacks in PHP applications but does not address the problem of user interactions. The Apollo tool (Artzi et al., 2010) does address user interactions but is restricted to PHP crashes or malformed HTML outputs. Such white-box testing tools combine concrete and symbolic (concolic) executions (Godefroid et al., 2008) by using a modified PHP virtual machine.

On the other hand, formal models and model-checkers have been used for test case generation since at least 1998 (Ammann et al., 1998), in a variety of contexts that has been surveyed elsewhere (Fraser et al., 2009). Most of this work concentrates on generating test cases that satisfy structural criteria on the model (e.g., state coverage, transition coverage, MC/DC coverage). As there is still no evidence of a strong relationship between such coverage criteria and fault detection effectiveness (Martin and Xie, 2007), we choose to rely on a domain-specific fault model. Our work is closely related to mutation testing (DeMillo et al., 1979; Jia and Harman, 2011). Even though mutation testing usually aims at assessing the effectiveness of a test suite to detect small syntactic changes introduced into a program, it can also be used to generate and not assess test cases. This idea was successfully applied for specification-based testing from AutoFocus, HLPSL or SMV models in the security context (Ammann et al., 2001; Büchler et al., 2011; Dadeau et al., 2011; Wimmel and Jürjens, 2002). Our work differs in that we start by real vulnerabilities in web applications and correlate them with specific mutation operators. Moreover, we do not stop after test generation but also provide a semi-automatic way to execute the generated test cases on real implementations, in our case, a web application.

More recently, Armando et al. (2011) have described work closely related to ours but for protocols instead of web applications. They start from an already insecure model described at the HTTP level and provide an automatic testing approach that relies on a mapping from each abstract HTTP element in the model to HTTP messages suitable for the SUV. The fully automatic procedure is achieved at the price of describing the model at the HTTP level. Even though this low level description is acceptable for protocols, we think it is more suitable to describe web applications at a higher abstraction and

[6]https://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf
[7]http://sectools.org

ask, if needed, the test expert to provide additional information during the testing procedure.

VI. Conclusion

We have presented a complete method for semi-automatically testing a Web application starting from a secure model. The first challenge was to find a way to use a model-checker for security testing although the model is secure. In order to get interesting attack traces from the model-checker, we introduce potential vulnerabilities in the model that are relevant for the System Under Validation (SUV), including Cross-Site Scripting (XSS) attacks. We analyzed known vulnerabilities in web applications to come up with corresponding mutation operators for models developed in ASLan++. This language was initially created for security protocols but is also suitable for web application specifications. We have seen that our mutation operators allow a model-checker to generate interesting attack traces.

The specification is modeled at a high abstraction level (i.e., abstract messages like `login`, `viewProfileOf` or `deleteProfileOf`, `editProfileOf`, `searchProfileOf`). Thus, even if our mutants are suitable for generating attack traces, the second challenge is to bridge the gap between the model abstraction and the real implementation to be able to (automatically) execute such attacks on the SUV. We presented a 2-step mapping for this purpose. The idea behind these two mappings is that the second one can be reused for testing other web applications. Thus, we introduced an abstract language (WAAL) to describe the attack traces at the browser level. The second mapping presented in this paper is suitable for translating actions described in WAAL to API calls using the Selenium testing framework.

Finally, the Test Execution Engine (TEE) automatically executes the attack trace when it is possible to reproduce it at the browser level. Otherwise, the TEE switches to the protocol level and may request the help of a test expert to execute an action, from the attack trace, at this lower level.

This methodology has been implemented and applied to widely popular WebGoat lessons. The high abstraction level for modeling the specification makes it easy to understand the secure model, without knowing the interactions at the protocol level between the web application and its users. Although the attack trace is described at this high level, the TEE is able to automatically reproduce most of it and for the rest, the intervention needed from a test expert is not very demanding as the TEE can guide him by providing some templates. We are aware that we have presented only a small preliminary set of experiments and we plan to conduct more experiments in the future. We are currently working on augmenting the library of mutation operators to represent the most common vulnerabilities in Web applications[8]. We are also working on extending our prototype to handle side-effect scripts, mainly for avoiding interference with the TEE.

[8] https://www.owasp.org/index.php/Top_10_2010-Main

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

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

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

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

A. Armando, D. Balzarotti, R. Carbone, A. Merlo, and G. Pellegrino, "From model-checking to automated testing of security protocols: Bridging the gap," 2011, submitted.

S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *TSE*, vol. 36, no. 4, pp. 474–494, 2010.

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

M. Büchler, J. Oudinet, and A. Pretschner, "Security mutants for property-based testing," in *TAP*, 2011, pp. 69–77.

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

R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Program Mutation: A New Approach to Program Testing," in *Infotech State of the Art Report, Software Testing*, 1979, pp. 107–126.

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.

G. Fraser, F. Wotawa, and P. Ammann, "Testing with model checkers: a survey," *STVR*, vol. 19, no. 3, pp. 215–261, 2009.

P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008, 16 pages.

W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation: tools and methodology," *STVR*, vol. 21, pp. 55–71, 2011.

W. Halfond, S. Anand, and A. Orso, "Precise interface identification to improve testing and analysis of web applications," in *ISSTA*, 2009, pp. 285–296.

Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, vol. 37, no. 5, pp. 649–678, 2011.

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.

E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *WWW*, 2007, pp. 667–676.

M. Turuani, "The CL-AtSe protocol analyser," in *RTA*, ser. LNCS, 2006, vol. 4098, pp. 277–286.

G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations," in *ICFEM*, 2002, pp. 471–482.

APPENDIX
ARCHITECTURE OF THE SPACITE TOOL

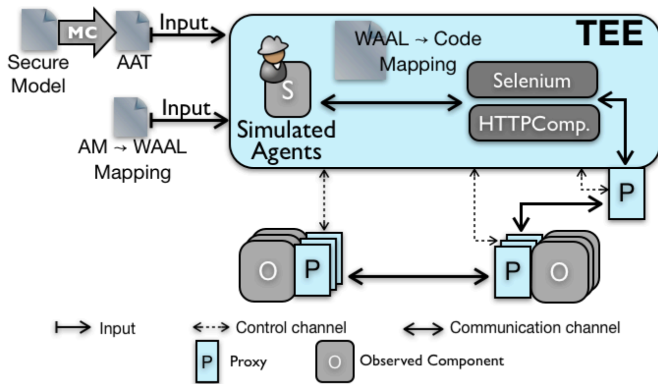

Figure 7. TEE architecture

The SPaCiTE tool takes as input a secure model described in ASLan++(AVANTSSAR, 2011). Even thought 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 the components at a high level (e.g., `login`, `viewProfile`, `updateProfile`), ignoring details about underlying protocols or communication information.

The SPaCiTE tool 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 web-application-dependent mapping, these high level messages are mapped to browser actions that generate these messages. Browser actions are parts of an intermediate language called WAAL. WAAL actions are mapped to executable code by an internal, framework-specific mapping. The first mapping is provided as an input to the tool. Finally this concrete test case is executed against the SUV by the TEE, described in Figure 7.

The SUV components are split into two distinct sets (i.e., 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 AAT. Observed components run in their normal environment except that their communication channels are monitored by proxies. Thus, the TEE can observe messages exchanged between observed components. An additional proxy is put in front of the TEE to intercept sent messages when the TEE is generating template messages for the test expert; these messages are not part of the AAT and therefore they must not reach any other component.