SPaCIoS

**Secure Provision and Consumption in the Internet of Services**

FP7-ICT-2009-5, ICT-2009.1.4 (Trustworthy ICT)

Project No. 257876

www.spacios.eu

---

# Deliverable D2.4.2
# Attacker behavior models and attack combination


## Abstract

Models of the behavior of the attacker, be they abstract or concrete, play a significant role in the test case generation technologies that are being developed in SPaCIoS. This deliverable presents two novel model-based testing approaches tailored to Web applications that are based on the concepts of attacker behavior models and attack combinations. Both techniques define attacker models and can be employed to test Web applications for complex vulnerabilities exploitable by combined attacks.


## Deliverable details

Deliverable version: *v1.0*
Classification: *public*
Date of delivery: *31.01.2014*
Due on: *31.01.2014*
Editors: *UNIVR*
Total pages: *50*

## Project details

Start date: *October 01, 2010*
Duration: *40 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INP, KIT/TUM, UNIGE, SAP, Siemens, IeAT

---

SEVENTH FRAMEWORK PROGRAMME

(this page intentionally left blank)

# Contents

# List of Figures

# List of Tables

# 1 Introduction

This deliverable addresses the problem of defining an attacker behavior model for model-based testing of Web applications, along with the definition of combinations of attacks.

**Motivations.** The definition of an attacker behavior model is a non-trivial task and, as already showed in [18], its definition tends to depend on the SUT (System Under Test). For example, while an attacker model à la Dolev-Yao is typically sufficient for the validation of security protocols, it does not seem to be appropriate for validating Web applications, as his cryptographic rules are of no help in this context (as showed in [18]).

Web applications can be complex structures (i.e., Service-Oriented Architectures) where known dangerous attacks, e.g. exploiting vulnerabilities reported in [13], can be easily avoided or confined. However, due to the underlying architectural complexity, even when the direct exploitation of given attacks has been avoided, there can be the possibility of exploiting a sequence of connected attacks (possibly not so dangerous when considered separately) leading to serious flaws. Consider, for example, a scenario in which a Web application is secure against CSRF attacks, i.e. it requires secret tokens to submit certain requests. Suppose that the same Web application is vulnerable to a social engineering attack allowing an intruder to obtain the secret token. In that case, the execution of the first attack (social engineering) allows for the exploitation of a second attack (CSRF) that would have not been possible to execute otherwise. Detecting such concatenations of exploits, to the best of our knowledge, is not possible with state-of-the-art tools. By using model-based testing, we are able to consider a wide and heterogeneous number of attacks, find connections between them (with respect to a particular scenario) and possibly detect chains of attacks that lead to complex flaws.

**Contribution.** We have formalized two distinct approaches. In Section 3, we propose a formal and flexible model-based security testing framework that supports a security analyst in carrying out security tests on SUTs. The approach is based on the use of the *components* of the SUT, where each component can be seen as an abstract representation of a functionality provided by the SUT that can be "used" through a user interface. Firstly, the security analyst defines a model of the SUT, specifying a proper set of components (either selected from a database or defined ad-hoc), the relation between them and a formalization of the security goals to be tested. A model checker is then used in order to generate counterexamples in the case when one or

more goals are violated: a counterexample specifies an "instance" of the system and the components for which the required security goal does not hold. Since the counterexamples are at a level of abstraction that does not allow for directly testing them on the SUT, a *concretization phase* is required. Such a phase relies on the definition, for each component, of a corresponding sequence of HTTP requests to be performed on the SUT and on the possible use of an *instantiation library* providing appropriate payloads. As a case study, a Web application presenting two vulnerabilities, an SQL-Injection and a Stored XSS, has been considered, thus showing how it is possible to use the framework in order to discover attacks based on the combination of multiple vulnerabilities.

As an alternative approach, in Section 4, we present a technique that aims at exploiting the relations between Web attacks. The intruder is defined by a set of attacks, formalized by following the typical paradigm of pre and post-conditions. Similarly, a set of initial conditions specifies the Web application and further sets of conditions are used to describe undesirable (goal) states of the system. A model checker is used to verify whether there exists any *attack chain* leading (by using the attacks as transition rules) from the initial state to a goal state. In this deliverable, we present an ASLan++ intruder model, based on a small set of significant Web attacks, and provide general guidelines for modeling Web applications. The methodology is applied to two examples, selected amongst SPaCIoS case studies.

**Structure of the deliverable.** The deliverable is structured as follows. Section 2 gives an overview of several connected approaches that we have used as a starting point for developing our techniques. In Section 3 and Section 4, formal descriptions of the two approaches are provided, together with reports of preliminary results. In Section 5, we conclude and propose some directions for future work.

# 2 State of the art

In this section, we give an overview of the main state-of-the-art techniques used for testing Web applications and for finding combinations of attacks. This makes more clear some of the theoretical choices we have made in Section 3 and Section 4, in particular with respect to the detection and usage of attack combinations.

The first paper [8] gives an overview of what state-of-the-art vulnerability scanners can and cannot detect on a Web application.

**Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners.** The authors of [8] present an evaluation of eleven black box Web vulnerability scanners. Black box Web vulnerability scanners are tools that automatically detect flaws in Web sites without, or with little, human support.

In order to test capabilities of different Web vulnerability scanners, the authors have implemented a flawed Web application (honeypot): WackoPicko[7]. Each tool has been challenged in both his crawling and vulnerabilities detection part. A detailed report section shows not only the results of the analysis but also a lot of information on the quality of the tools such as:

- false positives;
- number of accesses to Web pages;
- differences between manual, semi-automatic and automatic mode;
- list of detected vulnerabilities for each tool.

The conclusion of this paper highlights the need for more sophisticated crawling algorithms. In particular, it shows the impossibility of finding logical flaws of a Web application using the eleven tools considered in the paper. In fact, all the Web vulnerability scanners considered have failed to detect, and even check for, application specific vulnerabilities.

Model checkers, and formal analysis in general, have been widely used for the detection of logical flaws in (security) protocols and there are several approaches that aim to extend this idea also to Web applications. One first step in this direction has been done with [1].

**Towards a formal foundation of Web security.** In [1], the authors have presented a (formal) model-based method for the verification of Web applications. They propose a methodology for modeling Web applications and the results of the exploitation of the technique on five case studies, modeled in Alloy[10] modeling language.

A Web application model is composed by three main parts: Web concepts, threat models, and security goals. The first one represents the Web standards (e.g., HTTPS requests and responses), the second one is defined by an attacker model and the user behavior, that authors use to detect flaws in a Web application with respect to the security goals.

The Web concepts are:

- *Browser*, defined by three key pieces: a script context that represents all the scripts running in the browser, a security UI that is the browser's user interface used for security purpose, a state storage that contains information as cookies or saved passwords;
- *Servers*, modeled as network locations (abstraction of IP addresses);
- *Network*, that is the communication channel used by browser and servers.

For what concerns the threat models, the authors identify three different kinds of attacker and a user behavior:

- *Web attacker*: it controls Web servers, a number of DNS names and can obtain HTTPS certificates. It has no particular network privileges but can send HTTP requests to honest servers from network endpoints it controls. Once an honest agent visits an attacker endpoint, the attacker can access the user browser's APIs;
- *Network attacker*: it has all the abilities of a Web attacker and the ability to read, control, and block the contents of all unencrypted network traffic;
- *Gadget attacker*: it has all the abilities of a Web attacker as well as the ability to inject some limited kinds of content into honest Web sites;
- *User behavior*: it is needed to avoid spurious attacks (e.g., the user directly sends his credential to the attacker) and defines what a user can and cannot do, e.g., a user may visit any Web site but he does not distinguish between honest and dishonest endpoints.

The defined security goals are two:

- *Security invariants*: a set of invariants of the Web platform that have to remain true, e.g., user's browser will never generate a cross-origin HTTP request with DELETE method.
- *Session integrity*: when a server takes action based on receiving an HTTP request, the server wants to ensure that the request was generated by a trusted principal.

One drawback of the previous approach is that the authors have defined a general attacker model in order not to limit its behavior. The side effect is the high complexity of the scenarios that a model-checker has to handle. To solve this issue, the idea is to formally define a set of known Web attacks (more details in the following sections) in order to reduce the complexity of

the Web attacker without limiting the attacker behavior. There exist a lot of works in this direction, that instead of defining a set of general behavior rules for the attacker (as in [1]), define a set of vulnerabilities the attacker can search for. This can ultimately permit us to detect more complex attacks that rely on the combination of several attacks, exploiting more than one vulnerability. In the following we give a summary of four papers describing methodologies connected to this idea.

**Formal modeling of vulnerability.** In [9], the authors try to answer the question "Is it possible to construct a formal model of the domain of computer security vulnerabilities that is sufficiently robust to uncover meaningful relationships among multiple vulnerabilities that were not already recognized?". In order to do so, they present a framework to model vulnerabilities and their relationships. They take into account and model the vulnerabilities contained in the CERT Coordination Center Knowledgebase. Each vulnerability is modeled as a transition from a set of security relevant facts (preconditions) describing the state of the system, to a set of impacts (postconditions) representing what privileges and knowledge is gained by the intruder. They modeled 100 vulnerabilities (rules) and applied them to a set of facts representing the initial state of the system (collection of security facts including details such as program versions, operative system running on hosts, etc.). By modeling these artifacts in a rule-based production system, they were able to build dependency graphs representing how the exploitation of some vulnerabilities enables to exploit other ones, i.e. impacts of one vulnerability rule satisfies the preconditions of another vulnerability rule.

**A systematic approach to multi-stage network attack analysis.** In [4], a methodology for modeling and analyzing multi-stage network attacks is presented. The modeling phase concerns:

- a description of the *network*, in terms of physical topology, communication and trust relationships;
- a representation of the *attacker capabilities* over the elements of the network, e.g., to express that the attacker can communicate with a given host; and
- a list of *vulnerabilities*, expressed as pairs of preconditions, for the vulnerability to exist, and postconditions, describing the network state that can be reached by exploiting the vulnerability.

A formalism is presented in order to properly represent all these elements.

The models are analyzed by using an attack chaining algorithm, which takes as input a network state describing the initial conditions and outputs possible chains of attacks, produced by using the network topology and the attacker capabilities in order to exploit vulnerabilities. A prototype supporting attack tree generation is provided. Furthermore, the possibility of moving towards a quantitative analysis by using weights over the transitions, to be determined by a proper risk analysis, is discussed.

**Efficient generation of exploit dependency graph by customized attack modeling technique.** A more recent related work is [3]. Also this approach is based on building chains of attacks, here called *exploit dependency graphs*. The objective is to improve over traditional network security tools, which tend to leave out those attack paths obtained by interaction and interdependencies of vulnerabilities existing on different hosts. The method operates on conditions that represent attributes of network objects. Its input is represented by: a set of exploits (expressed as preconditions and postconditions), a list of hosts, the conditions of an initial state, the conditions of a goal state. The output is a dependency graph, containing chains of exploits possibly leading to a goal state.

The main distinctive features of the approach are the following:

- while the modeling phase of similar works is typically based on proprietary vulnerability databases, in this case the authors refer to open-source resources, mainly the *Metasploit* (`http://www.metasploit.com/`) database;
- a novel algorithm for backward search is proposed in order to retrieve the attack chains: its performances are compared with those of similar approaches and it is shown that the developed algorithm gives a significant improvement in terms of time and space complexity.

**Modeling Internet attacks.** A more low-level (i.e., vulnerabilities are linked to the implementation details of the architecture underlying the Web application) approach is presented in [20]. The authors aim to identify multi-stage attacks proposing, as they claim, a first scientific methodology.

Within the paper, an Internet attack model for capturing composite attacks is presented. The model is an extension of the attack tree concept in [16] mainly with parameters, precondition and postcondition assertions. A set of templates is provided and each template contains descriptive properties (such as a Mitre CVE database link), preconditions, subgoals, and postconditions. *Preconditions* express the system environment or configuration

properties, *subgoals* represent the antecedent objective of system intrusion, and *postconditions* identify state changes in the system and environments.

The authors present two different kinds of node in an attack tree: *concrete* and *abstract* nodes. While concrete nodes are identified by specific exploits, abstract nodes are used as placeholders in an attack tree for conceptual objectives such as system halts. For each node in the tree, the model also keeps track of the element of the system that is affected by the attack.

Another key aspect is the network model for which the authors define a specification language. Such a model, together with the attack tree model of the attacker, defines the model specification. Finally, an *attack visualization system* is described. It combines the parametric attack trees with a system specification language to support vulnerability assessment and attack visualization.

# 3 Components

*Penetration testing* [12, 19] is the most common approach for testing the security of Web applications, i.e., for generating a set of test cases (namely, pairs of inputs and expected outputs) and executing them on the *SUT* in order to acquire more confidence about the secure behavior of an application's implementation or to discover failures (i.e., unexpected behaviors). *Model-based testing* [5, 21] is not yet as widespread as penetration testing but it has been steadily maturing into a viable alternative/complementary approach.

Both these testing techniques, however, require quite some effort of the security analyst carrying out the tests, even when she may make use of existing tools, guidelines or libraries of common security vulnerabilities and attacks such as [15, 12]. In particular: penetration testing, which ranges from *black-box* to *white-box*, has helped uncover several vulnerabilities, but the experience of the security analyst carrying out the pen-tests is crucial for their success, especially in the case of black-box testing; in model-based testing, a formal model of the SUT is used to formally derive test cases, but this requires the security analyst to first create such a model, which may be quite a difficult endeavor especially in the industrial setting.

In this section, we propose a formal and flexible model-based security testing framework that supports a security analyst in carrying out security testing on SUTs. The main idea underlying this framework is that the use of model-checking techniques can automate the research of possible vulnerable entry points in the SUT, i.e., it permits an analyst to perform security testing without missing important checks. Moreover, the framework also allows for reuse: the analyst can collect her expertise into the framework and (re)use it

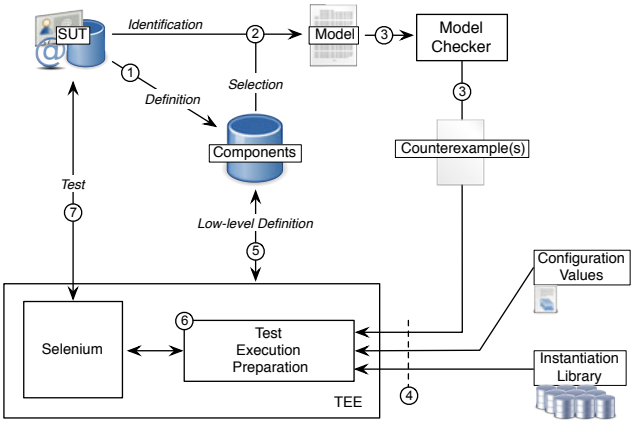


Figure 1: The proposed framework for model-based testing of Web applications (the arrows refer to interaction between elements or data passed between them, the numbers are used in the text for the explanation of the different phases).

during future tests on possibly different SUTs, which may be carried out by her or by members of the testing group of the analyst's organization, if any. In this way, the potentiality of a single test is not related to the expertise of the single analyst on a specific SUT but to the expertise of the entire testing group.

## 3.1 A framework for model-based security testing of Web applications

Our framework (depicted in Figure 1 and described in detail in Section 3.1.1) uses the components of the SUT both in the modeling of the application (Section 3.2) and in the concretization of the test cases (Section 3.3). Intuitively, we define a component as an abstract representation of a functionality provided by the SUT that can be "used" through a user interface (e.g., a Web browser). *Using* a component means that it is possible to perform a sequence of ordered HTTP requests leading the user to access a functionality provided by the component.

Web applications offer various types of functionalities, ranging from general purpose functionalities such as authentication, editing of private information or searching information, to specific functionalities such as reading a newsfeed or purchasing goods from an online shop. During the modeling phase, we are not interested in all the implementation details of a single component in order to consume it; rather, we focus our attention on the interaction between components themselves and between components and the

data they have access to. For example, we specify that the authentication component requires credentials in order to authenticate a user and that administrative actions such as modifying the personal profile can be performed only after the component for authentication (i.e., the editing profile component comes after the authentication component).

### 3.1.1 The phases of our framework

The first thing that a security analyst has to do when using our framework is to define components from the SUT (phase ① in Figure 1). During this phase, the security analyst has also to check, manually, if some of the existing components in the database can be (re)used in order to model the SUT (if two applications share the same functionality, then she can reuse the associated component), and, if not, she has to insert into the database the new component(s) (see Section 3.2 for further information about the definition of component). With the database populated with a proper set of components, the security analyst has the means to create the *model* of the SUT (②). The model includes a selected subset of the defined components (identified with respect to the SUT to be tested), the relation between them, and a specification of the security goals to be tested.

The model is then passed to a model checker (③). Our framework is general and thus it is not bound to a specific model-checking tool; for concreteness we employ the Alloy Analyzer [10], which takes a model, its constraints and its security goals written in the Alloy syntax, checks the goals in the model and generates counterexamples if one or more goals are violated, i.e., a counterexample shows for what instances of the system and for what components the specified security goal does not hold.

The fact that the counterexamples are indeed abstract gives, however, rise to two problems (related to the level of abstraction) that we have to tackle.

First, the counterexamples are at a level of abstraction that does not permit us to directly test them on the SUT, since it specifies the components used to violate the goal but not how these components should be used in the real implementation. The framework thus provides for a *concretization phase*, which relies on the fact that we can define for each component a corresponding sequence of HTTP requests to perform on the SUT. The definition of the relations between components and HTTP requests (⑤) is performed through SeleniumHQ IDE [17] (in the following, simply "Selenium") by the security analyst. Each component must be "recorded" on the SUT and the corresponding sequence of actions generating HTTP requests is saved as an XML file that will be used during the execution of the test cases.

Second, the counterexample specifies which components have to be used, but its level of abstraction does not allow for the specification of attack-dependent data. If we have to use a specific payload, the *Instantiation Library* provides it. The Instantiation Library contains data such as attack strings (e.g., payloads for XSS), common malicious input (e.g., a set of passwords for a brute force attack) and scripts to be used as test patterns (i.e., script to be executed client-side in order to test the SUT). We will see later how the entries collected in the library are also saved with the information that bind them to specific component(s) that they attack or compromise.

The final phase of our framework envisages a TEE, a semi-automatic test execution technology with which the security analyst can interact in order to drive the execution of the test cases on the SUT. The TEE provides a connection with the Instantiation Library and the data, contained in the *Configuration Values*, needed for the interaction with the SUT (④), and takes care of "translating" (via the *Low-level Definition* ⑤) the counterexample(s) (④) into executable test cases. At the end of this phase, the information contained in the counterexample(s), the components and the HTTP requests are thus combined in the creation of a suite of test cases (⑥) that we can run using Selenium (⑦).

We will now describe these phases in more detail.

## 3.2 Modeling Web applications for security testing

In order to model Web applications,

- the components, data types and properties are used to describe the behavior and the features of the SUT;
- the relations between components are used to assign a specific data type or a property to components, and to define a preorder between components;
- the security goals represent the vulnerabilities for which the security analyst wants to test the SUT.

In Figure 2, we show a simple example of a two-phase authentication mechanism that requires the user to provide personal credentials, in the first phase, and a token, in the second phase (this type of multi-level login is common in home banking Web applications, for instance). We use this scenario as an example of the use of components during the modeling of Web applications (① and ②).

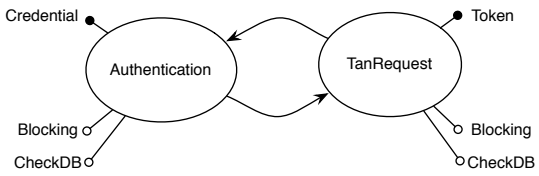


Figure 2: An example of the usage of components in a multi-level-login scenario. The empty circles represent properties of the components and the filled circles represent the data types handled by the components.

### 3.2.1 Defining components, data types, properties, and relations between components

In our framework, a component is an abstract representation of a functionality provided by an SUT through parts of a user interface.

From the security analyst's perspective, the definition of components (①) is a simple task during which she has to identify the functionalities of interest, i.e., the functionalities that she wants to test with respect to some security goals. That is, we assume that the security analyst is able to divide a given SUT in $i$ functionalities each represented by a component $c_i$, so that we can define the set of components $Comp = \bigcup_i c_i$.[1] As an example, for the SUT in Figure 2, we have two components:

$$Comp = \{Authentication, TanRequest\}.$$

Let $Req_i$ be a single HTTP request and $HTTP = \{Req_1, \ldots, Req_n\}$ an ordered set of HTTP requests. Since every component $c_i$ is "connected" to an SUT, we also assume the existence of a usability function

$$\mu \colon c_i \to HTTP \tag{1}$$

that maps each component to a sequence of HTTP requests. Even though the usability function will not be used in the modeling phase, it can be used to determine if we are defining proper components: in the case it is not possible to determine a proper set of HTTP requests, we have a hint that we are not representing a component.

Every functionality given in an SUT must handle some type of data; in order to represent the nature of these data without modeling their values, we define $d_a$ as a data type handled by components and $Data = \bigcup_a d_a$ as the set of all the data used by the SUT.

[1] We assume that the security analyst is able to exploit her experience to identify these $i$ functionalities alongside the specification of the SUT. Similar assumptions are

In order to give to the security analyst the freedom to define properties or activities on which she can base her analysis (i.e., define a security goal), we define the set $Prop = \bigcup_x p_x$ where $p_x$ is a specific property.

The data types and the properties associated with a component are used to abstract the model from the implementation without losing the possibility of performing targeted attacks against a functionality (expressed by the component). The granularity of these two sets can change the abstraction level of the model; at this time, we are not focused on defining the right level of abstraction, and we do not define a particular granularity for the sets *Data* and *Prop*. Changing the granularity also changes the expressiveness of the model and, in particular, of the class of security goals that we can formalize. We only give basic examples of the use of these two sets, without going into the details of this granularity.

For the SUT in Figure 2, the data that the authentication mechanism handles are:

$$Data = \{Credential, Token\},$$

where we chose to model the pair *login* and *password* as the data type *Credential* because in this particular model we are not interested in the single input but only in the concept of "secret" information that the user supplies to the Web application, and to model the *Token* as a Transaction Authentication Number (TAN). The properties are:

$$Prop = \{Blocking, CheckDB\},$$

where *Blocking* represents the fact that the component is meant to forbid the access to the components that follow (we will see the need for this property in the next section), until the data associated with the component (i.e., credentials) are provided, and *CheckDB* represents that the data associated to that component are used to query a database.

It is not enough to define the sets *Comp*, *Data* and *Prop* containing the information needed to describe the structure and the data of the SUT. We need to define also the relations on these three sets to make sure that the final model behaves the way that we intuitively expect.

We assume the existence of three relations $\lesssim_c$, $\mathcal{R}_D$ and $\mathcal{R}_P$ between components, between components and data types, and between components and properties, respectively. With a slight abuse of notation, we use these relations to create the corresponding sets of pairs that define them for a given

commonplace in software engineering.

model:

$$\lesssim_c = \{(c_i, c_j) \mid c_i, c_j \in Comp \wedge c_i \lesssim_c c_j\} \quad (2)$$
$$\mathcal{R}_D = \{(c_i, d_a) \mid c_i \in Comp \wedge d_a \in Data \wedge c_i \,\mathcal{R}_D\, d_a\} \quad (3)$$
$$\mathcal{R}_P = \{(c_i, p_x) \mid c_i \in Comp \wedge p_x \in Prop \wedge c_i \,\mathcal{R}_P\, p_x\} \quad (4)$$

These relations represent

(2): a preorder $c_1 \lesssim_c c_2$ between components, meaning that $c_1$ should be used before $c_2$;

(3): a non-empty relation $c_i \mathcal{R}_D X$ where $X \subseteq Data$, meaning that $c_i$ handles the data types in $X$;

(4): a non-empty relation $c_i \mathcal{R}_P Y$ where $Y \subseteq Prop$, meaning that $c_i$ has the properties in $Y$.

When the model checker searches for counterexamples, it must discard those that violate any of these relations.

Referring again to the example in Figure 2, the corresponding sets are easy to define:

$$\begin{aligned}
\lesssim_c &= \{(Authentication, TanRequest), (TanRequest, Authentication)\} \\
\mathcal{R}_D &= \{(Authentication, Credential), (TanRequest, Token)\} \\
\mathcal{R}_P &= \{(Authentication, Blocking), (Authentication, CheckDB), \\
&\quad (TanRequest, Blocking), (TanRequest, CheckDB)\}
\end{aligned}$$

Summing up, let $Req_i$ be a single HTTP request, $HTTP = \{Req_1, \ldots, Req_n\}$ an ordered set of HTTP requests, $Comp$ a set of components, $Data$ a set of data types $\{d_0, \ldots, d_n\}$ handled by components, $Prop$ a set of properties $\{p_0, \ldots, p_m\}$, and $\mu$ a usability function that maps a component to the HTTP requests that it generates. Then we can finally define:

**Definition 1** A *component* $c_i \in Comp$ is an abstract representation of a functionality that is provided by an SUT through parts of a user interface on which we can define the relations (1), (2), (3) and (4).

### 3.2.2 Defining security goals

From Definition 1, we have the means to define the security goals that the model should satisfy. The purpose of security goals is to verify that some properties or conditions hold on the model. Our framework allows us to perform vulnerability assessments on SUTs. We restrict our attention to those

security goals that represent well-known vulnerabilities, but our framework is open to considering other, more complex or less standard, vulnerabilities.

In general, in order to exploit a vulnerability, some conditions related to the data and some properties, must be fulfilled. We formalize informal conditions related to a vulnerability by means of a formula over the three sets defined in (2), (3) and (4). For example, a *Stored XSS* [11] occurs when the attacker saves some data (usually a malicious client-side script) on the server, and then these data are returned to other users in the course of regular browsing. If we assume $\mathit{WriteDB}, \mathit{ReadDB} \in \mathit{Prop}$, then we can formalize that the security analyst wants to verify the absence of a Stored XSS on the SUT by means of the formula

$$\begin{aligned}&\forall\, c_1 \in \mathit{Comp}\,.\, \forall\, c_2 \in \mathit{Comp}\,.\\ &(\ (\ \exists\, d \in \mathit{Data}\ .\ (c_1, d) \in \mathcal{R}_D \wedge (c_2, d) \in \mathcal{R}_D\ ) \rightarrow\\ &(c_1, \mathit{WriteDB}) \notin \mathcal{R}_P \vee (c_2, \mathit{ReadDB}) \notin \mathcal{R}_P \vee (c_1, c_2) \notin \lesssim_c^*)).\end{aligned}$$

Intuitively, this assertion says that we may not have a Stored XSS when foreach pair of components ($c_1$, $c_2$), if there exists a data type handled by both the components, then it must be that $c_1$ does not write, or $c_2$ does not read, or $c_2$ does not come before $c_1$.

As another example consider a vulnerability for bypassing the authentication mechanism of a Web application (e.g. SQL-Injection or password brute forcing). Such a vulnerability can be used by an attacker to access a Web application without prior knowledge of a valid credential. Assume $\mathit{CheckDB}, \mathit{Blocking} \in \mathit{Prop}$. An easy assertion for verifying if there is a component that could be vulnerable to authentication bypassing, can be expressed (if the formula is true) as follows:

$$\forall\, c \in \mathit{Comp}\,.\, (c, \mathit{CheckDB}) \notin \mathcal{R}_P \ \vee\ (c, \mathit{Blocking}) \notin \mathcal{R}_P\,.$$

This assertion says that we may have an authentication bypassing vulnerability if the data associated to a component is used to query a database and that component provides authentication.

The assertions presented in this section are not bound to a specific SUT, they only depend on the *Data* and the *Prop* of the components. The security analyst can thus create a set of security goals to be used during her tests without the need of (re)writing them every time.

## 3.3 A concretization methodology: from high to low level

In Section 3.2, we have seen how we can use components in order to model an SUT; we will now see how components can be bound to HTTP requests

through a concretization methodology. The purpose of this methodology is to specify for each component how it is used as a sequence of HTTP requests in order to let the TEE execute the counterexample (see Section 3.4.3 and Figure 4 for an actual counterexample) on the SUT. As a concrete example, we chose to use Selenium for the definition of components as HTTP requests but, of course, other equivalent technologies could be used. Selenium is a Firefox plugin that allows one to record, edit, and debug tests. A *test* is a recorded sequence of actions that a user can perform through the interface of the SUT; the tests can then be exported as reusable scripts that can be later executed (see [17] for further information about Selenium).

Selenium is allowed to record only actions that can be performed through a Web browser (e.g., clicking a button, selecting from a drop-down menu), which means that we have to limit our investigation only to the browser interface of the SUT. We are aware that, in this way, we are missing a family of attacks (e.g., tampering HTTP messages), but in this deliverable we want to give a proof of concept of our methodology and the use of Selenium is well fitted to the definition/recording of the components that we are testing. We will consider the missing attacks in future work.

The Selenium features that we want to use are (i) the recording of browser actions in order to generate HTTP requests and (ii) the possibility to replicate these actions. Our concretization phase relies on the fact that we ask the security analyst to record through Selenium the set of browser actions that compose each component. Selenium records the actions performed through a Web browser automatically during the interaction of the security analyst with the application.[2] The data acquired during the recording phase can be saved by Selenium in different formats, including the XML format that we chose here. In the XML file, we can find information like:

| | |
|---|---|
| open | open a target link in the Web browser |
| select | select target label from a drop down menu |
| type | type a text in a target text field |
| clickAndWait | click a target object and wait for the application response |

In our framework, the security analyst must record a Selenium XML file for each component (⑤); when she is recording these files, she must: (i) perform the browser actions only on the browser, and (ii) provide inputs, if it is required by the actions. The recording of these actions is not a difficult task. To do so, we envision three possible moments (related to three testing scenarios):

[2]In Selenium, a set of low-level actions recorded in one session is called a *test case*; in this deliverable, we reserve this terminology for a set of low-level actions created during the creation of the test suite.

Table 1: Components and corresponding low-level actions recorded using Selenium. (a) generates one HTTP request whereas (b) generates two HTTP requests.

(a) *login*

| **Action** | **Target** | **Data** |
|---|---|---|
| open | page.php | |
| type | usernameForm | bob |
| type | passwordForm | password |
| clickAndWait | signinButton | |

(b) *manageProfile*

| **Action** | **Target** | **Data** |
|---|---|---|
| open | page2.php | |
| select | optionsMenu | label=bob |
| clickAndWait | button | |
| type | Name | Bob |
| type | Surname | Paulson |
| type | Phone | 123456 |
| clickAndWait | button | |

- after the definition of all the components in the SUT (①), if the SUT must be tested exhaustively and the security analyst wants to create different models of the application;
- after the definition of the model (②), if the security analyst wants to test only the functionalities described in the model;
- after the generation of the counterexample (③), if the security analyst does not want to record all the components but only the ones contained in the counterexample.

We show an example of components recorded with Selenium in Table 1: each component is associated to a set of actions that can be performed through a Web browser generating one or more HTTP requests. In Section 3.4, we will use these same low-level actions in order to concretize our counterexample(s).

The last phase of our concretization methodology is managed by the TEE. This component uses the Selenium records to translate the counterexample into an executable format. It also instantiates the attack (using the Configuration Values and the Instantiation Library) and uses Selenium to execute the test on the SUT. Table 1 shows that the actions performed through the Web browser are saved by Selenium along with the values that they require.[3] In order to make the concretization of a counterexample independent from the data inserted during the recording of the components, the TEE removes from the XML code all these values.

We are now ready to generate the concrete test cases, a task that is carried out by the TEE sub-component *Test Execution Preparation* (⑥ in Figure 1).

[3]In Selenium, it is not possible to record an action without providing the required

The last piece of information needed are the input fields data that we have removed after the recording of the components. In order to generate a set of executable test cases (a test suite), the TEE uses the data contained in:

- the Instantiation Library, to select the payload to use during the attacks (e.g., a malicious script), and in
- a file of Configuration Values, containing the data known before the attack related to the SUT (e.g., a valid username and password).

The payloads are stored in the Instantiation Library along with the corresponding data types and component properties that they attack or compromise, so the TEE can create specific test cases for each component in the counterexample (④). In the last step, the TEE executes the test suite on the SUT (⑦) using Selenium.

The TEE has two possible execution modes conforming to the testing methodology selected by the security analyst:

- in *semi-automatic* mode, the analyst will be asked to provide an input for the missing field(s) (with the possibility of selecting it from the Instantiation Library);
- in *automatic* mode, the input will be chosen only from the Instantiation Library and the Configuration Values; the TEE will use the information associated to each payload in order to continue the execution.

In both cases (as happens in our case study), the TEE or the security analyst can select from the Instantiation Library a malicious payload.

If every action is executed without any errors from Selenium or the application (in the case an action cannot be executed), the TEE reports the execution trace as a successful attack; otherwise, an error is returned and the TEE asks the security analyst how to proceed.

## 3.4 A detailed case study

In order to assess the strength of our framework, we have implemented a vulnerable Web application called *Chained Attack*, which presents two vulnerabilities: a SQL-Injection and a Stored XSS. This case study thus also shows how our framework is able to generate tests for the discovery of a single attack even if the combination of multiple vulnerabilities is needed.

input.

### 3.4.1 Scenario

We have developed *Chained Attack* as a typical Web application where the user starts interacting with a log-in page in which he has to provide his credentials; if the credentials are correct, the user can select which profile he wants to view in an ad-hoc page. Moreover, the application provides users with the functionality to modify their own profiles.

We consider a scenario where a security analyst wants to verify that a user's profile cannot be used to perform a Stored XSS attack, i.e., the profile viewed by a legitimate user cannot be injected with malicious content from an attacker. Under this assumption, the test cases are generated for an attacker with complete access to the application (as provided by the Selenium recording).

### 3.4.2 Modeling the *Chained Attack* Web application

As remarked in Section 3.1.1, in the concrete instantiation of our framework that we consider as an example, we use the Alloy formal language to model the relations between the components, the data types they consume, the properties related to their behavior, and the security goals.

**Components** In our framework, each Alloy model start with the definition of the abstract signatures for components, the data types they consume and the properties related to their behavior. The relations in the component's abstract signature directly reflect Definition 1: (i) `succ` represents the preorder between components (equation (2)), (ii) `dataTypes` represents the data that the component handles (equation (3)), (iii) `properties` represents the properties of the component (equation (4)). The abstract signatures of `dataType` and `property` are empty because we do not need to express further relations:

```
abstract sig Component{
   succ : set Component,
   dataTypes: set DataType,
   properties: set Property  }

abstract sig DataType{}

abstract sig Property{}
```

**Components Model** The first phase in the modeling of SUTs adopted in our framework (① in Figure 1) consists in the identification of the components so as to use them to create the model (② in Figure 1). In order to

formalize the three sets *Comp*, *Data* and *Prop* in Alloy, we use *signatures*, which express the vocabulary of a model. In our example, we have identified four components: `Login`, `SelectProfile`, `EditProfile` and `ViewProfile`. In addition, we expand the model with a `Start` component to identify the starting state(s) of the SUT. For each component, we have identified the data types it handles and properties related to its behavior.

The four components are defined by the following code:

```
one sig Start, Login, SelectProfile, ViewProfile,
        EditProfile extends Component {}
```

With regards to the data types, we decided to define a single signature for the pair *username, password* named `Credential`, the type `Text` for one (or more) field(s) handled by the components, and the type `Id` refers to an identification value used to select a profile:

```
one sig Credential, Text, Id extends DataType{}
```

With regards to the properties, `Blocking` represents the fact that a component is meant to forbid the use of the following components until proper credentials are provided, `Check` represent that the data associated to that component is used to query a database, and `ReadDB/WrinteDB` represent that the data associated to that component is read/write from a database:

```
one sig Blocking, ReadDB, WriteDB, CheckDB extends Property{}
```

**Decorated Components Model** In Alloy, a fact expresses a statement about the model that must remain true during the search for counterexamples. In the model, we use facts to formalize the relations between components, data types and properties (i.e., signatures). In the following, we give the Alloy code that defines the relations depicted in Figure 3.

- Definition of the preorder relation:

```
fact {succ = Start->Login +
    Login->SelectProfile +
    SelectProfile->ViewProfile +
    SelectProfile->EditProfile +
    ViewProfile->EditProfile +
    EditProfile->ViewProfile +
    Login->Login + EditProfile->EditProfile +
    ViewProfile->ViewProfile +
    EditProfile->SelectProfile +
    ViewProfile->SelectProfile }
```

- Definition of the data types handled by the components:

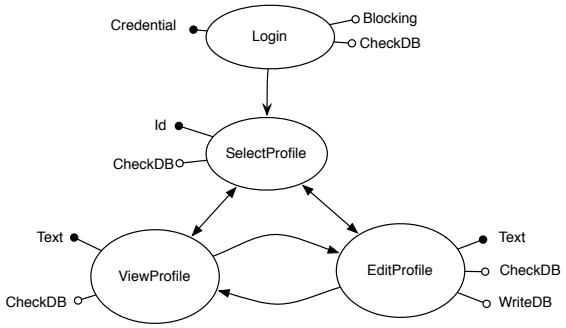


Figure 3: Graphical depiction of the model representing the components of the *Chained Attack* SUT. We show the relations among component, data types and properties of components.

```
fact {dataTypes = Login->Credential +
    SelectProfile->Id +
    ViewProfile->Text +
    EditProfile->Text }
```

- Definition of the properties of each component:

```
fact {properties = Login->Blocking +
    Login->CheckDB +
    SelectProfile->ReadDB +
    ViewProfile->ReadDB +
    EditProfile->ReadDB +
    EditProfile->WriteDB }
```

The model is depicted in Figure 3.

**Security goals** Security goals are modeled in Alloy as assertions, which are properties or conditions that must be true during the evolution of the model. While a fact is used to force something to be true of the model, an assertion is a claim that something must already be true due to the rest of the model. We model the assertion that allows us to identify those components that may be involved in a Stored XSS as follows (see Section 3.2.2 for an explanation on how to define security goals in our framework).

The following assertion formalizes the conditions under which a Stored XSS is more likely to happen (i.e., all the possible scenarios where a data is written on the database and then viewed by a user). That is to say, there is not the possibility of performing a Stored XSS if there does not exist a component `c1` such that, for some component `c2`, `c1` performs a write and `c2` performs a read of the same data types and `c2` comes after `c1`. The last two

lines of the `assert` refer to the fact that the second component (`c2`) must read the data after the first component (`c1`) has written it:

```
assert StoredXSS {
     no c1 : Component |
     some c2 : Component |
     WriteDB in c1.properties &&
     ReadDB in c2.properties &&
     c1.dataTypes=c2.dataTypes &&
     c2 in last.current &&
     c1 in last.*prev.current }
```

Assertions are verified (or falsified) using a `check` statement; in our example, we used

```
check StoredXSS for 5 Component, 3 DataType,
     3 Property, 5 ApplicationState
```

which means that the model checker will search for a counterexample comprised of five components, three data types, three properties, and bounded to five states of the application execution. For what concerns this bound, note that the analysis of the model in our framework typically starts from the lowest number of states, which is usually the number of component signatures; thus, the check statement above could not be used on more complex models or different assertions. If no counterexample is found, the application state bound should be increased by the security analysis as much as she sees fit.

**State evolution** We define the signature `ApplicationState` in order to keep track of the current component that is used in the application (only one component can be executed in a particular state). We then create a linear (total) ordering over `ApplicationState` as follows:

```
sig ApplicationState { current : one Component }
open util/ordering[ApplicationState]
```

`Start` is the initial state from which the execution of the components starts:

```
fact { first.current=Start}
```

As final step we define how the model changes the current component (following the `succ` relation) when the application changes its state. In other words, for every couple of adjacent states, the relation current is changed to one of the components pointed by the relation `succ`:

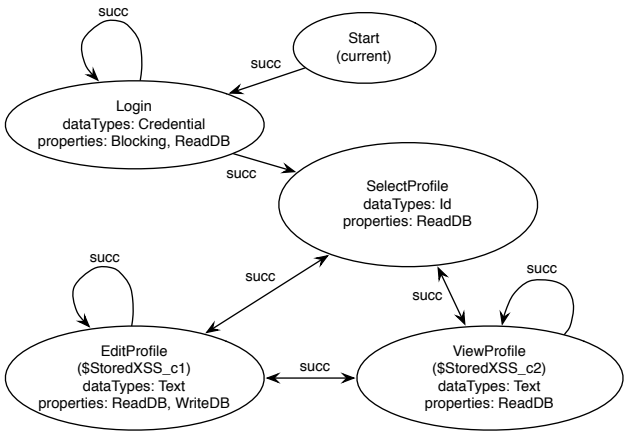


Figure 4: Graphical representation of the counterexample reporting the presence of a possible Stored XSS in the *Chained Attack* case study. The Alloy Analyzer reports the two components involved in the attack: `$storedXSS_c1` as the writing component and `$storedXSS_c2` as the viewing component.

```
fact {
  all s: ApplicationState, s': s.next{
      currentMove [s.current, s'.current] }
}
pred currentMove[curr,curr': one Component] {
  one x: curr | one y: x.succ | {
       curr' = curr - x + y }
}
```

### 3.4.3 Counterexample

At the end of the modeling phase, we have a formal model of the SUT, which is then given as input to the model checker in order to be formally validated. In case of a vulnerability, the model checker returns a counterexample containing the ordered sequence of component executions that violate the goal (the sequence is obtained through the preorder between components). In Figure 4, we give the counterexample for the model with respect to the security goal given in Section 3.4.2.

The counterexample generated by the model checker is not directly executable on the implementation. It does not provide information about the vulnerabilities that can be exploited, it just warns that there exist some components of the application that may lead to a Stored XSS. The next framework phases are needed to express how components have to be executed on the SUT and what inputs have to be used.

### 3.4.4 Concretizing the attack

**Components and low-level actions** As explained in Section 3.3, in order to concretize a counterexample, the security analyst has to define the relation between components and HTTP requests. In this concrete instance of our framework, and thus in our case study, we use Selenium to bridge the gap between the abstract representation and the actual implementation of the SUT. Each component extracted from the SUT has to be recorded with Selenium (Section 3.3).

From the interaction with the SUT, we can also extract information about the correctness of the model defined in the previous phase(s). Since we can define the relation between components and Selenium actions after the specification of the model or after the generation of the counterexample, the interaction with the SUT can give a basic validation of the model. If it is not possible to define a Selenium recording, it means that the security analyst has made a mistake and attempted to define a component without respecting Definition 1.

**Building the test suite** The output of the previous phases is twofold: a counterexample that violates the assertion and a set of Selenium actions for each component. The counterexample is not directly executable on the SUT, and we have to deal with the problem of filling the abstraction gap between the counterexample and the implementation of the SUT.

In our framework, the concretization phase is carried out automatically by a test suite creation technology that: (1) translates each component contained in the counterexample in the corresponding sequence of Selenium actions and (2) specify the missing attack-dependent data using the *Instantiation Library* (that contains common malicious input) and the *Configuration Values* (that contain data known before the attack). The data in the Instantiation Library are also used to generate multiple test cases from a single counterexample.

This information is processed by the TEE during the execution of the tests.

Once the previous phases have been completed, the TEE takes care of executing the test on the SUT. Following the test case generated, it first has to use the `Login` component. According to the selected testing methodology (as explained in Section 3.3), the TEE asks the security analyst to provide proper input, or tries to bypass the `Login` component injecting a payload. In the latter case, since the username "Bob" is in the Configuration Values, the TEE first checks the Instantiation Library for a payload that has `Credential` as `Data type` and `Blocking` as `Component properties`, as

Table 2: Example of Instantiation Library showing the association between data types, component properties and payloads for the *Chained Attack* SUT.

| Data types | Component properties | Attack type | Payload |
|---|---|---|---|
| Credential | CheckDB, Blocking | Auth. Bypassing | `' or '1'='1' or '1'='1 root'-` |
| Text | WriteDB | XSS | `<script>alert('xss');</script> <IMG SRC=javascript:alert('XSS');> '';!-" <XSS>=&{()}` |
| Credential | CheckDB, Blocking | Auth. Bypassing | `Administrator'- ' HAVING 1=1 -` |

shown in Table 2. The table also shows that the TEE selects the payload `' or '1'='1'` corresponding to a SQL-Injection, and uses it in the password field of the Selenium recording of the `Login` component.

At this point, the TEE automatically accesses Bob's session and asks what input should be used to interact with the `SelectProfile` component. As before, the security analyst can rely on the Instantiation Library to try to inject malicious payload or she can decide to select a particular profile. If she decides to select Bob's profile, the TEE asks what input should be used for the `EditProfile` component. Once again, the security analyst can rely on the Instantiation Library to try to inject malicious JavaScript code that will be stored in that profile. The TEE then uses the `ViewProfile` component in order to complete the attack, resulting in the execution of the injected malicious code. Table 3 shows the complete test case generated by the TEE.

Such an attack could be exploited by an attacker as a chained composition of a SQL-Injection and a Stored XSS attacks. That is, from the attacker's prospective in a real attack:

- the attacker exploits the SQL-Injection vulnerability to bypass the log-in page and thus impersonate a legitimate user,
- the profile of the target user is modified by the attacker with the injection of malicious JavaScript code (the target user is in the Configuration Values),
- a legitimate user logs-in and views the profile of the compromised user, and
- the legitimate user renders the compromised profile and executes, for example, the stored JavaScript code.

Table 3: Test case generated automatically by the TEE from the Instantiation Library and the Configuration Values for the *Chained Attack* SUT. The horizontal lines separate the three components extracted by the TEE from the counterexample.

| **Action** | **Target** | **Data** |
|---|---|---|
| open | index.php | |
| type | id=signin-username | Bob |
| type | id=signin-password | `' or '1'='1'` |
| clickAndWait | id=button-signin | |
| open | dashboard.php | |
| select | id=options | label=bob |
| clickAndWait | css=button.button | |
| open | profile.php | |
| type | name=name | Bob `"><script>alert('xss');</script>` |
| type | name=surname | Paulson |
| type | name=phone | 123456 |
| clickAndWait | css=button.button | |
| open | dashboard.php | |
| select | id=options | label=bob |
| clickAndWait | css=button.button | |

With this case study, we have shown how in our framework the security analysis can (1) the identify components, (2) relate them to vulnerabilities of interest, and (3) leverage on these vulnerabilities during the execution of the tests.

# 4 Chained Attacks

In this section, we describe a technique, named *Chained Attacks*, for model-based testing of Web applications, based on the analysis of relations between Web attacks. The idea (depicted in Figure 5) is to create an ASLan++ model of a Web application, define a goal (i.e., attack state) we want to reach and, via model checking, obtain an abstract attack trace that leads a Web application to the goal state. The intruder is defined by a set of attack formalizations; that is, a set of Web attacks (e.g., CSRF, XSS ...) is formalized in ASLan++ so that the intruder can use them in order to attack a Web application. More specifically, the Web attacker we define in Section 4.1 is composed by a set of rules (i.e., Web attacks) defined with the usual paradigm of pre and of post-conditions. The Web application is defined by a set of initial conditions and the goal is also a set of conditions but describing an undesirable state of the system. A model checker, by starting

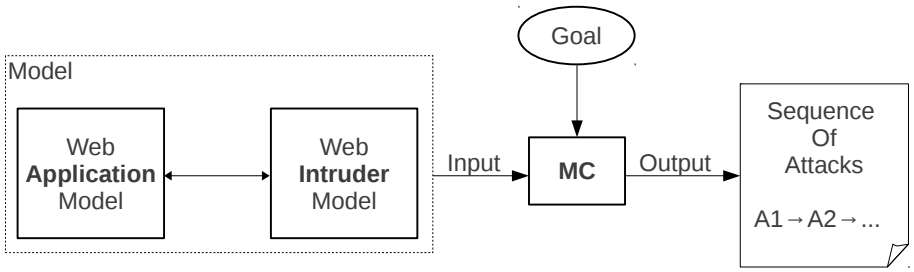


Figure 5: Chained attack overview

from a state where the initial conditions hold and by using the transition rules of the Web intruder (describing the attacks), looks for attack traces leading to the goal state.

As showed in Section 2, some similar approaches have been developed. Here, we aim to gather the ideas underlying those approaches and adapt them to the context of the SPaCIoS project; in particular, we re-define (using ASLan++ as a language) and extend the attacks presented in related works (see Section 2), in particular those of [9]. Note also that we avoid those approaches that aim to use an attacker à la Dolev-Yao (DY). We also aim to create a set of guidelines for modeling Web applications, so that the level of abstraction is appropriate to their use by a model checker (in contrast with what happens in [1]) and to the Web attacker we define.

The idea of defining a Web intruder that uses a set of Web attacks to test the security of a Web application is justified by the fact that typically (i) attacks on Web applications exploit more than one vulnerability, (ii) state-of-the-art tools do not search for attack relations and (iii) the DY intruder is "too strong" for testing Web applications: in fact, the report in [18] shows that the level of abstraction of a DY intruder is not appropriate for validating Web applications, as, e.g., his cryptographic rules are of no help in this context.

We support the use of our methodology by means of two examples (Section 4.3.3).

## 4.1 Intruder model for chained attacks

The intruder model we employ differs from the Dolev-Yao intruder [6] which is built in the AVANTSSAR tools because, as described in detail in Section 4.2, we are not considering models capturing the message exchange between clients and the Web application servers. In fact, we do not consider the malicious actions that an intruder may perform on messages. We just consider under which circumstances an intruder may attack a Web application

and what are the consequences of carrying out those attacks.

For this purpose, we model attacks to Web applications in ASLan++ by means of a `select{on(COND):{CONS}}` branch. The guard `COND` contains all the facts representing the preconditions for the execution of the attack, e.g. to execute an SQL Injection attack there must be a query built by employing some kind of input submitted by a user of the Web application. If a state satisfies such conditions, then the intruder may execute the corresponding attack resulting in a change of state characterized by `{CONS}`. This block of statements models the consequences, e.g. disclosure of information, unauthorized access, ..., of the attack performed.

All the formalized attacks are gathered together into a specification file named `Attack_specifications.aslan++` under a `while(true)` loop. The loop allows for the concurrent execution of every `select` branch whenever their `COND` guards are satisfied by the current state of the model of the Web application. This means that an intruder can execute any attack whenever the preconditions for its execution formalized in the guard `COND` hold.

The collection of all these attack steps represents our Web application attacker model as it provides the model-checkers with all the possible attacks that can be performed against such kind of applications. As already mentioned, for convenience and reusability purposes, all attack branches are contained in an ASLan++ file named `Attack_specifications.aslan++`. Such file contains also all the definitions necessary to model the attacks, such as function facts, variable symbols and Horn Clauses introducing facts corresponding to the execution of some attacks, e.g. in Listing 1 the execution of a `sql_injection_unauthorized_access` attack corresponds to bypassing the restrictions put in place by a form, i.e. knowing a secret such as a password, by a dishonest user who does not know (possess) the required secrets (privileges). It is not necessary to model also what is the action performed upon submitting the form because, accordingly to the modeling guidelines described in Section 4.2, that aspect will be defined into the Web application model. This modeling choice allows us to define a general intruder model which can be used with every Web application model.

Listing 1: Example of Horn Clause in the intruder model specification file

```
sql_injection_unauthorized_access_hc(User, Form, Db):
  action_form(User, Form, Db) :-
    sql_injection_unauthorized_access(action_form(User, Form, Db));
```

In fact, one of our goals is to have a Web attacker model general enough to be reused for Web application specifications created by modelers following our guidelines. To do so, they just need to import the `Attack_specifications.aslan++` file in the model of the Web application they want to analyze.

We remark that the current definition of the attacker model (Section 4.3.1) contains only a few attack definitions, aimed at applying the methodology to the examples of Section 4.3. A more comprehensive definition of the attacker model is left for future work.

## 4.2 Modeling guidelines

In order to reuse the Web attacker model described in Section 4.1 and to obtain attack traces representing chained attacks, we present here a set of modeling guidelines that modelers have to follow when writing the models of a Web application.

First of all, the modeler has to import the intruder model contained in the `Attack_specifications.aslan++` file, by using the statement `import Attack_specifications;`. This will automatically provide the modeler with all the symbols defined in the intruder model file[4]. A list of function symbols that have to be used to define essential aspects of the Web application follows:

```
% list of facts used to define conditions for attacks,
% these same facts has to be used by the modeler to model
% the web application as it follows:

% to specify that an agent has the rights to access a page
grant(agent, page): fact;

% to bound a page to the form(s) it contains
inputForm(page, form): fact;

% to model who submitted a form and which db this may affect
action_form(agent, form, info set): fact;

% to model who submitted a form without storing something on a db
search_form(agent, form): fact;

% to model the fact that input(s) of a form is used in a query
% executed on a db
execQuery(form, info set): fact;

% to specify that input(s) of a form is stored into a db
storing(form, info set): fact;

% to specify that input(s) of a form is saved into a variable and then
    printed
printValue(form): fact;
```

Starting with this core set of function symbols[5], we now describe the set of

[4]Note that the `Attack_specifications.aslan++` file must be in the same folder of the Web application model it is imported into, or the `-DASLANPATH` option has to be used to set the path where the file is stored

[5]Every other symbol (variables or functions) we are going to mention in the rules, which has not already defined in the attacker model, has to be defined in the `symbols` section of the Web application model

rules composing the guidelines for modeling Web applications in the context of our work on chained attacks.

**Rule 1: form's action.** Every form is characterized by an action that is performed upon submission of certain inputs. We already mentioned the `action_form` symbol (`action_form(agent, form, info set): fact;`), which is used to model the fact that a user has been able to execute an action through the submission of a form. The last argument, of type `info set`, represents the set of information that is targeted by the action triggered by the form. Note that using an ASLan++ set type does not mean that the form is used to access a database. It is just an abstraction used to represent all kinds of containers, e.g. databases, files, memory, and many others.

The `action_form` fact itself does not tell anything about the consequences of executing the action related to the form. As already mentioned, this is due to the fact that we use `action_form` also in the intruder model in the `Attack_specifications.aslan++` file. Therefore, the modeler has to write suitable Horn Clauses using `action_form` in the body to introduce facts describing what a form is actually doing. For this purpose, we propose a template of Horn Clause which will be triggered every time a form is executed:

```
<form_name_constant>_<action>_hc(User):
  <consequence_fact> :-
    action_form(User, <form_name_constant>, <db>)
```

where: `<form_name_constant>` is the constant symbol of type `form` that has to be defined by the modeler to represent the form; `<action>` is just a name used to identify what is being modeled by the clause; `User` is a variable of type `agent`; `<consequence_fact>` is the fact produced by the Horn Clause and modeling the consequences of executing the action associated to the considered form; and `<db>` is the name of the `info set` that the action is affecting/depending on.

**Rule 2: public pages.** The modeler has to identify the publicly accessible pages of the application. Those pages may contain some forms, e.g. to access a restricted area upon showing some sort of secret, and therefore all users must be allowed to view those pages and their content. To model this, the `grant` function fact symbol has to be used. Since there may be a large number of users, it is probably faster to use Horn Clauses rather than manually introducing every single `grant(user, page)` fact in the `body` section. For this reason, the modeler may prefer to use the following template:

```
grant_all_<page_constant>_hc(User):
  grant(User, <page_constant>) :-
    true;
```

where `<page_constant>` is the constant symbol defined for the publicly accessible page and `User` is a variable of type `agent`. Note that the body of the clause consists of the `true` fact which is automatically defined by the ASLan++-Connector during the translation from ASLan++ to ASLan, and that is always present in the initial state defined in the `section inits` in the ASLan specification.

**Rule 3: Web application structure.** After defining what happens upon executing each form and which pages are publicly accessible, the modeler has to provide essential information about the structure of the Web application. These information are provided in the body of the single entity present in the Web application model in form of fact assertions and has to cover all the following aspects:

- *Relation between pages and forms:* this bound is given by means of the `inputForm(page, form)` function fact. It is sufficient to define constant symbols for every page and form and then instantiate the necessary facts and assert them in the body.
- *Technical details appearing as preconditions in the attack definitions:* for the time being, the only aspects that need to be modeled are which of the forms stores or prints the information provided as input, and which form triggers an action involving the execution of a query. The former detail can be modeled by using the function fact `storing( form, info set)` or `printValue(form)`, and the latter by means of the `execQuery(form, info set)`.

All these rules have been applied to write the specification for the WebGoat application scenario. Further details and examples on how to instantiate the rules described in this section can be found in the remainder of this document, and in particular in Section 4.3.2 where the model of the "Injection Flaws LAB Stage 1: String SQL Injection" lesson is described.

## 4.3 WebGoat SQL Injection Lesson

We have used the WebGoat[14] project as a case study in order to put into practice our modeling methodology. WebGoat is an interactive learning Web application from OWASP, the Open Web Application Security Project which is structured in lessons. In each lesson, users must demonstrate their understanding of a security issue by exploiting a real vulnerability in the WebGoat applications. Typically, each lesson is designed to test the user for one specific vulnerability. However, some lessons allow to perform a different attack

besides the one the lesson was meant for. For example, the "Injection Flaws LAB Stage 1: String SQL Injection" lesson is affected by a SQL-Injection vulnerability which can be exploited to bypass the login phase. However, once the authentication page has been bypassed, it is possible to exploit also a Reflected XSSin the search profile page. In this section we show how we have modeled this particular lesson in ASLan++ using the attacker model proposed in Section 4.1 and the guidelines in Section 4.2.

### 4.3.1 The Web application attacker

In Section 4.1 we have presented the general idea to represent the behavior of a Web intruder in terms of the attacks it can perform. We stated that the intruder model will be represented in the form of preconditions (i.e. something that must be true in order to exploit a vulnerability), and consequences (i.e. something that will be true once the vulnerability has been exploited). At this stage, we have focused our attention in the modeling of just three attacks: SQL-Injection and two variants of XSS, i.e. Stored and Reflected XSS. However, these attacks are too general to be considered useful when modeling a specific attack. In fact, in the presence of a SQL-Injection vulnerability, it is possible to compromise a system in many different ways. According to the CAPEC [2] definition, a SQL-Injection can lead to arbitrary queries being executed, causing disclosure of information, unauthorized access, privilege escalation and possibly system compromise. The compromised scenario, as well as the state of the attacker, depends on the payload of the SQL-Injection. For example, a SQL-Injection attack whose payload is meant to bypass a login page will enable the attacker to get access to a restricted area he was not previously authorized to enter. In the case the payload is meant to access restricted information stored in the database, the attacker will increase his knowledge but will not have direct access to a restricted area of the Web application. The same thing can be said for XSS attacks. As for SQL-Injection attacks, in this case the outcome of an attack depends on the payload of the script submitted. For this reason, we let the representation of SQL-Injection and XSS attacks depend on the payload possibly employed.

We now consider each vulnerability and how it has been modeled as preconditions:

- **SQL-Injection (unauthorized access):** we considered the case in which it is more likely that a SQL-Injection vulnerability is used to get access to a restricted area. In order to exploit this scenario, a dishonest user has to get access to a page in which there is an input form that is used to compose a SQL query. In Listing 2 we have reported the

`on():{}` statement related to a SQL-Injection attack for bypassing an authentication mechanism.

- **Stored/Reflected XSS (alert box):** we considered the standard case in which a Stored/Reflected XSS attack is used to display an alert box to the user. In particular, we focus on the case in which an attacker is looking for an XSS within an HTML input form. The submission of the form generates a request where the URL contains the XSS payload. In order to exploit this scenario (for the Stored XSS) a dishonest user has to get access to a page in which there is an HTML input form. Once submitted, the value of the form is stored somewhere inside the Web application itself[6]. In Listing 3 we have reported the statements modeling the XSS attacks.

Listing 2: SQL-Injection's preconditions for bypassing authentication mechanism

```
% SQL_Injection (bypass check on credentials)
on( grant(?User,?Page) &
  dishonest(?User) &
  inputForm(?Page,?Form) &
  execQuery(?Form,?Db)
):{
  sql_injection_unauthorized_access(action_form(User, Form, Db));
}
```

Listing 3: XSS attacks preconditions.

```
% Stored XSS for alert box
on( grant(?User,?Page) &
  dishonest(?User) &
  inputForm(?Page,?Form) &
  storing(?Form,?Db)
):{
  stored_XSS(action_form(User, Form, Db));
}

% Reflected XSS
on( grant(?User,?Page) &
  dishonest(?User) &
  inputForm(?Page,?Form) &
  printValue(?Form)
):{
  reflected_XSS(search_form(User, Form));
}
```

Listing 4: Horn Clauses modeling actions of the payload of Stored and Reflected XSS attack.

[6]We assume that each stored input is, eventually, used to generate part of the content

```
stored_XSS_alert(User, Form, Db):
  stored_XSS_alert :-
    stored_XSS(action_form(User, Form, Db));

reflected_XSS_alert(User, Form):
  reflected_XSS_alert :-
    reflected_XSS(search_form(User, Form));
```

Listing 5: Horn Clauses modeling actions of the payload of the SQL-Injection attack.

```
sql_injection_unauthorized_access_hc(User, Form, Db):
  action_form(User, Form, Db) :-
    sql_injection_unauthorized_access(action_form(User, Form, Db));
```

Within the body of each attack statement there is a fact symbol related to a particular payload. Once the preconditions are met, the related fact is introduced. The facts are used in the attacker's clauses in order to represent a specific behavior for a specific payload. For example, consider the `stored_XSS` fact. Since we are modeling a Stored XSS attack in which the payload is only the displaying of an alert box, as it is in the case of reflected XSS, we have created a clause representing this behavior (Listing 4).

In order to create the attacker model as general as possible, we use a fact symbol, i.e. `action_form`, that expresses the consequences of executing a form when a form is part of the preconditions. In this way, the security analyst has only to model the behavior of each form of the application and the attacker will automatically adapt to the modeled scenario. For example, consider the `sql_injection_unauthorized_access` symbol. In this case, the payload allows the attacker to successfully execute the action associated to that form. Thus, the relative clause is nothing more than the action for that form (Listing 5).

### 4.3.2 The Scenario

The scenario represents the behavior of the Web application that has to be tested. As described in Section 4.2, we want to achieve a modeling process that does not necessarily require the modeler to be a security expert. The effort of the modeler is to observe the scene and describe the way it works. Consider the "Injection Flaws LAB Stage 1: String SQL Injection" lesson. The user is presented with a login page within a login form used to access a restricted area of the Web application. Once the user has provided valid credentials, he has the right to access a select page. This page contains a form allowing the user to fire a request for viewing the details of the selected profile. When viewing the profile, the user has also the ability of editing the

of a dynamic page.

details via an edit page. The edit page contains an editing form that fires a request for storing the information within the Web application itself.

To model this scenario with ASLan++ (Listing 6) we use clauses for modeling the consequences related to the execution of a form, and we use basic symbols to model relation between pages, forms and the behavior of each form. The complete model of this scenario is in Section A.1.

Listing 6: ASLan++ code of the "Injection Flaws LAB Stage 1: String SQL Injection" lesson.

```
clauses

  % login_page: executing the login gives access to the
      select_profile_page
  login_page_grant_hc(User):
    grant(User, select_profile_page) :-
      action_form(User, login_form, login_db);

  % and it gives access to search profiles page too
  grant_search_profile_page_hc(User):
    grant(User, search_profile_page) :-
      action_form(User, login_form, login_db);

  % select_profile_page: selecting one profile gives access to the
      view_edit_page
  select_profile_page_grant_hc(User):
    grant(User, view_edit_page) :-
      action_form(User, view_form,  profile_db);

  % view_edit_page: editing a profile results in viewing the updated
      profile
  % thus it gives access to the same page
  view_profile_page_hc(User):
    grant(User, view_edit_page) :-
      action_form(User, edit_profile_form, profile_db);

  % the login_page is publicly accessible
  grant_all_home_hc(User):
    grant(User, login_page) :-
      true;

body {

  % 1) correlation between pages and forms
  inputForm(login_page, login_form);
  inputForm(search_profile_page, search_profile_form);
  inputForm(select_profile_page, view_form);
  inputForm(view_edit_page, edit_profile_form);

  % 2) which of the forms stores the information provided as input
  storing(edit_profile_form, profile_db);

  % 3) which of the form fields is visualized in a web page
  printValue(search_profile_form);

  % 4) which of the forms trigger the execution of a query on which db
  execQuery(login_form, login_db);
  execQuery(view_form, profile_db);
```

```
}
```

### 4.3.3 Experimental results

We have checked for both reflected and stored XSS using CL-AtSe model checker. It has returned two abstract attack traces, finding two chained attacks, one for each vulnerability.

**Reflected AAT.** The trace is composed by four steps:

- *step 0.*

```
0 % (WG_SQLi,19)
  CLAUSES: {  }
  RULES:     step_001_WG_SQLi__line_178(root,dummy_set_info,dummy_form
      ,0,n19(IID_1),dummy_page,dummy_agent)
```

  The first step is an initialization step in which the rule of the WebGoat lesson entity (representing the modeled scenario) is executed.

- *step 1.*

```
1 % (WG_SQLi,22)
  CLAUSES: { grant_all_home_hc(i) }
  RULES:     step_002_WA_Attacker__line_139(root,dummy_set_info,
      login_db,dummy_form,login_form,0,dummy_page,login_page,10,
      dummy_agent,i,dummy_agent_1,n19(IID_1))
```

  With step 1, the Web attacker accesses the login page (that is publicly available).

- *step 2.*

```
2 % (WG_SQLi,28)
  CLAUSES: { grant_search_profile_page_hc(i),
             sql_injection_unauthorized_access_hc(login_db,login_form,
                 i) }
  RULES:     step_004_WA_Attacker__line_162(root,dummy_set_info,
      dummy_form,search_profile_form,0,dummy_page,search_profile_page
      ,10,dummy_agent,i,dummy_agent_1,n19(IID_1))
```

  The Web attacker, by using a SQL-Injection, bypasses the login page and gets access to the search profile page.

- *step 3.*

```
3 % Goal failure
  CLAUSES: { reflected_XSS_alert(search_profile_form,i) }
```

  Finally, the Web attacker performs a reflected XSS from the search profile form.

**Stored AAT.** The trace, similarly to the previous one, is composed by four steps:

- *step 0.*

```
0 % (WG_SQLi,19)
  CLAUSES: {  }
  RULES:     step_001_WG_SQLi__line_178(root,dummy_set_info,dummy_form
      ,0,n19(IID_1),dummy_page,dummy_agent)
```

  The first step is an initialization step in which the rule of the WebGoat lesson entity (representing the modeled scenario) is executed.

- *step 1.*

```
1 % (WG_SQLi,22)
  CLAUSES: { grant_all_home_hc(i) }
  RULES:     step_002_WA_Attacker__line_139(root,dummy_set_info,
      login_db,dummy_form,login_form,0,dummy_page,login_page,10,
      dummy_agent,i,dummy_agent_1,n19(IID_1))
```

  With step 1, the Web attacker access the login page (that is publicly available).

- *step 2.*

```
2 % (WG_SQLi,28)
  CLAUSES: { grant_search_profile_page_hc(i),
             sql_injection_unauthorized_access_hc(login_db,login_form,
                 i) }
  RULES:     step_004_WA_Attacker__line_162(root,dummy_set_info,
      dummy_form,search_profile_form,0,dummy_page,search_profile_page
      ,10,dummy_agent,i,dummy_agent_1,n19(IID_1))
```

  The Web attacker, by using a SQL-Injection bypasses the login page, gains access to the search profile page.

- *step 3.*

```
7 % Goal failure
  CLAUSES: { stored_XSS_alert(profile_db,edit_profile_form,i) }
```

  Finally, the Web attacker performs a stored XSS from the search profile form.

**AATs Concretization** In order to test the two AAT reported from CL-AtSe we have manually tested both the traces on the WebGoat application. Only the reflected AAT has been successfully executed because a sanitization algorithm has blocked the insertion of malicious code while updating the profile of users for the stored AAT.

As depicted in Figure 6 we have performed five steps in order to concretize the reflected AAT. The first frame of the figure is the login page of the

WebGoat lesson in which we have chosen the admin username. Following the AAT we have bypassed the login phase (as showed in the second frame) using a SQL-Injection (namely, `'or'a'='a`) and we have gained access to the user list page. We clicked on the "search staff" button and, as showed in the fourth frame, instead of writing the name of a user, we have written a Javascript code. Once we have clicked on the search button, a reflected XSS has been performed as showed in the last frame.

## 4.4 Discussion and future work

In this section we presented a technique, called *Chained Attacks*, for model-based testing of Web applications, based on the analysis of relations between Web attacks. Our approach follows the ones illustrated in Section 2 as we model the intruder by using a set of transitions. Every transition represents an attack to a Web application and it is executed whenever the preconditions for performing the attack hold. Once it has been executed, the transition drives the system into a state where the consequences of the attack are introduced and this may lead to subsequent attacks. In contrast to the existing techniques, we focused on the context of Web application presenting the first prototypes of Web attacks formalized into the ASLan++ file `Attack_specifications.aslan++` (available in Section A.2).

In addition to this, we also presented a set of modeling guidelines for creating ASLan++ specifications modeling Web applications in such a way that the Web attacker model in `Attack_specifications.aslan++` can be employed by simply importing it.

To show the viability of our technique, we applied it to a case study, the "Injection Flaws LAB Stage 1: String SQL Injection" WebGoat lesson. We illustrated how to apply the rules defined in Section 4.2 and showed the result of the model-checking phase: the attack traces representing two chained attacks, i.e. Reflected and Stored AAT. We also discussed the concretization of such AATs, explaining why it was possible to execute the Reflected AAT and not the Stored one.

The final goal of this work is to obtain a set of attack traces representing chained attacks to Web applications in order to automatically build an attack tree, similarly to what is presented in [3, 4]. In order to achieve this, we are going to improve our prototype specification of the Web attacker by adding further transitions, so as to capture all the best-known attacks that an intruder may perform against Web applications. At the same time, we plan to employ other model-checking tools in order to find the one that is more suited to generate the complete set of attack traces and that works better with our attacker model defined in the ASLan++ file `Attack_specifications.aslan++`

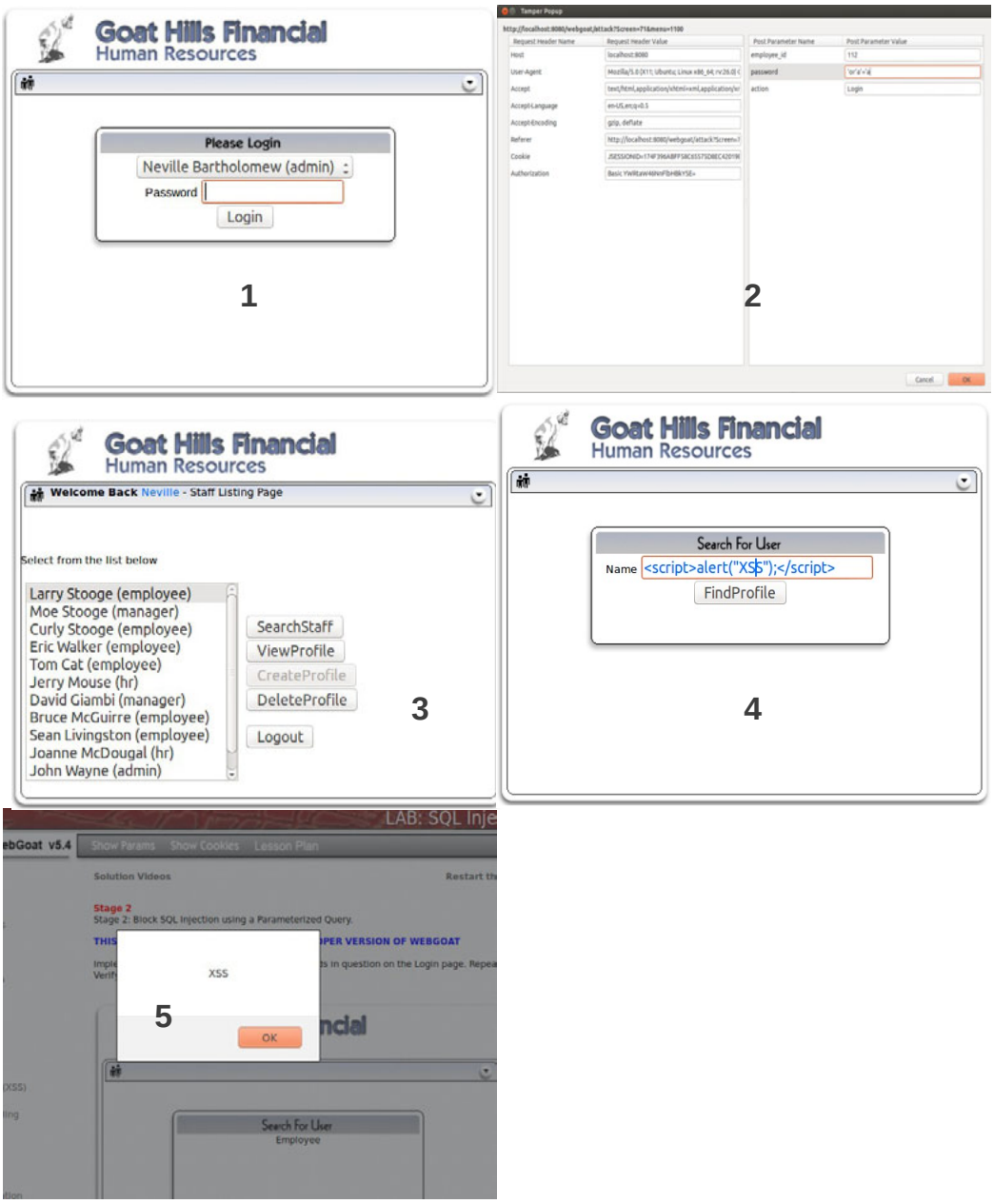


Figure 6: WebGoat, SQL-Injection (bypass login) followed by Reflected XSS

. To this purpose, we might use a prototype version of SATMC which returns in output all the attack traces for the analyzed model, but we might as well try to use tools outside the AVANTSSAR Platform. In fact, AVANTSSAR's tools employ a built in Dolev-Yao attacker that, because of the presence of our Web attacker model, we do not require. Even if using model-checkers not employing the Dolev-Yao attacker (which is very powerful as it controls the network and can do almost everything with the messages circulating on it) would be probably better in terms of computational time needed to check our models, a deeper analysis is required in order to determine whether such an improvement justifies the effort (and outweighs the drawbacks) of replacing ASLan++ with the specification languages used by other tools.

# 5 Conclusions

In this deliverable, we have described two approaches for model-based testing of Web applications, with a particular emphasis on the detection of chains of attacks. On one hand, both the approaches rely on the common idea of presenting a methodology for the formal specification of the SUT in order to use a model-checker for detecting possible flaws. On the other hand, the two approaches differ in several aspects: the one described in Section 3 provides a framework for security testing while the approach in Section 4 is mainly devoted at presenting a novel attacker model. Furthermore, they use different technologies and they require different levels of detail for what concerns the input specification.

One of the strengths of the *Components* approach (Section 3) is the reusability of the components being defined, as well as of the sets *Data* and *Prop* (for these sets, the expertise required for populating the Instantiation Library is automatically "reused"). The reusability of components has also an implicit impact on scalability: the security analyst can identify existing parameters for the SUT from a set of known components (with associated parameters) or she can define new ones and then improve the set of components. If this feature may sound trivial for the more skilled penetration tester, we believe that the adoption of the framework by a testing group can result in a significant improvement for the testing capability of the whole group. New attack techniques and payloads can be inserted into the framework without modifying (or compromising) the testing methodology. For what concerns the security goals, we are aware that a basic knowledge of logic is required in order to write them down, but their reusability compensates the efforts put in their definition. In fact, as stated before, the security goals are a high-level representation of vulnerabilities, and as such they can be reused

by the security analyst when dealing with different SUTs.

The *Chained Attacks* approach (Section 4) aims at adapting several state-of-the-art methodologies (Section 2), based on the idea of studying the combination of attacks of a different sort, to the context of the SPaCIoS project. It has been designed with the intent of reusing SPaCIoS results and technology as far as possible. In fact, tools developed inside the SPaCIoS project have been used for both the implementation of an attacker model and the validation of case studies. In terms of scalability, the attacker model proposed can be easily imported in any ASLan++ specification (provided that the modeler follows the guidelines of Section 4.2 for formalizing the Web application) and this makes the proposed methodology of practical interest, in particular, for all those approaches that use ASLan++ as a specification language.

# References

[1] D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304, 2010.

[2] CAPEC. *CAPEC – Common Attack Pattern Enumeration and Classification, release 2.6*. The MITRE Corporation, 2013. Available at http://capec.mitre.org/.

[3] Ishan Chokshi, Nirnay Ghosh, and Soumya K. Ghosh. Efficient generation of exploit dependency graph by customized attack modeling technique. *18th International Conference on Advanced Computing and Communications (ADCOM)*, 0:39–45, 2012.

[4] Jerald Dawkins and John Hale. A systematic approach to multi-stage network attack analysis. In *Proceedings of the 2nd International Information Assurance Workshop (IWIA)*, pages 48–58, 2004.

[5] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech '07*, pages 31–36. ACM, 2007.

[6] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, IT-29(2):198–208, 1983.

[7] Adam Doupé. Wackopicko vulnerable website, 2010. Available at https://github.com/adamdoupe/WackoPicko.

[8] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and*

*Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 111–131. Springer Berlin Heidelberg, 2010.

[9] William L Fithen, Shawn V Hernan, Paul F O'Rourke, and David A Shinberg. Formal Modeling of Vulnerability. *Bell Labs Technical Journal*, 8(4):173–186, 2004.

[10] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

[11] OWASP. Cross-Site Scripting, www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[12] OWASP. OWASP Testing Guide v3, www.owasp.org/index.php/OWASP_Testing_Project.

[13] OWASP. *OWASP Vulnerability categories*. OWASP, 2011. Available at https://www.owasp.org/index.php/Category:Vulnerability.

[14] OWASP. OWASP WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2011.

[15] The Nmap Project. Top 125 Network Security Tools. www.sectools.org.

[16] Bruce Schneier. Attack trees. http://www.schneier.com/paper-attacktrees-ddj-ft.html, december 1999.

[17] SeleniumHQ: Web Application Testing System http://seleniumhq.org/.

[18] SPaCIoS. Deliverable 2.5.1: Framework for Concretisation of Abstract Tests, 2013.

[19] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons, Inc., 2011.

[20] Terry Tidwell, Ryan Larson, Kenneth Fitch, and John Hale. Modeling internet attacks. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and security*, volume 59, 2001.

[21] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

# A ASLan++ models for chained attacks

## A.1 Web Goat SQL-Injection lesson

Listing 7: Complete model from the WG_SQLi.aslan++ file

```
1 % Web Goat lesson "Injection Flaws LAB Stage 1: String SQL Injection"
```

```
2  % @modeler(Alberto Calvi, UniVr, 2013)
3  % @modeler(Federico De Meo, UniVr 2013)
4  % @modeler(Marco Rocchetto, UniVr, 2013)
5
6  specification WG_SQLi channel_model CCM
7
8  entity WG_SQLi {
9
10   % importing attacker steps
11   import WA_Attacker;
12
13   symbols
14
15     % Here the modeler has to define the costant symbols for every
16     % web page, form and databases of the web application
17     login_page, select_profile_page, view_edit_page, search_profile_page:
           page;
18     login_form, view_form, edit_profile_form, search_profile_form: form;
19
20     % the following costant symbol of type "info set" are defined
21     % for every page that has a set of saved information associated
22     % e.g. login_db contains the credentials that users has to show
23     % to log into the web application. This does not mean that there is
24     % actually a DB, there may be also other ways to store such information
25     login_db,profile_db: info set;
26
27   clauses
28
29     % Horn Clauses satisfying rule 1, i.e. form action
30
31     % login_page: executing the login gives access to the
           select_profile_page
32     login_page_grant_hc(User):
33       grant(User, select_profile_page) :-
34         action_form(User, login_form, login_db);
35
36     % and it gives access to search profiles page too
37     grant_search_profile_page_hc(User):
38       grant(User, search_profile_page) :-
39         action_form(User, login_form, login_db);
40
41     % select_profile_page: selecting one profile gives access to the
           view_edit_page
42     select_profile_page_grant_hc(User):
43       grant(User, view_edit_page) :-
44         action_form(User, view_form,  profile_db);
45
46     % view_edit_page: editing a profile results in viewing the updated
           profile
47     % thus it gives access to the same page
48     view_profile_page_hc(User):
49       grant(User, view_edit_page) :-
50         action_form(User, edit_profile_form, profile_db);
51
52     % Horn Clauses satisfying rule 2, i.e. Public pages
53
54     % the login_page is publicly accessible
55     grant_all_home_hc(User):
56       grant(User, login_page) :-
57         true;
58
59   body {
```

```
60
61      % Here the modeler has to assert all the facts composing the
62      % initial state of the web application, including:
63
64      % 1) correlation between pages and forms
65      inputForm(login_page, login_form);
66      inputForm(search_profile_page, search_profile_form);
67      inputForm(select_profile_page, view_form);
68      inputForm(view_edit_page, edit_profile_form);
69
70      % 2) which of the forms stores the information provided as input
71      storing(edit_profile_form, profile_db);
72
73      % 3) which of the form fields is visualized in a web page
74      printValue(search_profile_form);
75
76      % 4) which of the forms trigger the execution of a query on which db
77      execQuery(login_form, login_db);
78      execQuery(view_form, profile_db);
79    }
80
81    goals
82
83      not_XSS: [](!reflected_XSS_alert);
84  }
```

## A.2 The Web attacker model

Listing 8: Complete model from the WA_Attacker.aslan++ file

```
1   % Web Application attacker for chained attacks
2   % @modeler(Alberto Calvi, UniVr, 2013)
3   % @modeler(Federico De Meo, UniVr 2013)
4   % @modeler(Marco Rocchetto, UniVr, 2013)
5
6   % @atse{remember to set an adequate number for the --nb option}
7
8   % specification WA_Attacker channel_model CCM
9
10  entity WA_Attacker{
11
12    types
13
14      page < message;
15      info < message;
16      form < message;
17
18    symbols
19
20      % variable symbols
21      User: agent;
22      Page: page;
23      Db: info set;
24      Form: form;
25
26      % this fact can be used by the modeler to build terms of type "info"
27      info(message): info;
28
29      % list of facts used to define conditions for attacks,
30      % these same facts has to be used by the modeler to model
31      % the web application as it follows:
```

```
32
33      % to specify that an agent has the rights to access a page
34      grant(agent, page): fact;
35
36      % to bound a page to the form(s) it contains
37      inputForm(page, form): fact;
38
39      % to model who submitted a form and which db this may affect
40      action_form(agent, form, info set): fact;
41
42      % to model who submitted a form without storing something on a db
43      search_form(agent, form): fact;
44
45      % to model the fact that input(s) of a form is used in a query
46      % executed on a db
47      execQuery(form, info set): fact;
48
49      % to specify that input(s) of a form is stored into a db
50      storing(form, info set): fact;
51
52      % to specify that input(s) of a form is saved into a variable and then
            printed
53      printValue(form): fact;
54
55      % list of facts used to specify that an attack occurred and
56      % to trigger HCs that will produce the action on the web application
57      % resulting from the execution of the attack, e.g. an "action_form" fact
58      % indicating that a form has been submitted and the corresponding
59      % action has been performed (probably with malicious input(s))
60      sql_injection_unauthorized_access(fact): fact;
61      stored_XSS(fact): fact;
62      reflected_XSS(fact): fact;
63
64      % fact used in HC as flag indicating that stored_XSS occurred
65      % this is just used for defining a goal in the model WG_SQLi
66      stored_XSS_alert: fact;
67      reflected_XSS_alert: fact;
68
69    clauses
70
71      % HCs that will produce the action on the web application
72      % resulting from the execution of the attack. This action could
73      % be either the ones performed by forms (action_form(...)) or other
74      % strictly dependent on the attack performed, e.g gaining access
75      % to the content of a dataset/database
76
77      sql_injection_unauthorized_access_hc(User, Form, Db):
78        action_form(User, Form, Db) :-
79          sql_injection_unauthorized_access(action_form(User, Form, Db));
80
81      stored_XSS_alert(User, Form, Db):
82        stored_XSS_alert :-
83          stored_XSS(action_form(User, Form, Db));
84
85      reflected_XSS_alert(User, Form):
86        reflected_XSS_alert :-
87          reflected_XSS(search_form(User, Form));
88
89    body {
90
91      while(true){
92        % all the attacks are modeled as "select on" branches
```

```
93        select{
94          % SQL_Injection (bypass check on credentials)
95          on( grant(?User,?Page) &
96            dishonest(?User) &
97            inputForm(?Page,?Form) &
98            execQuery(?Form,?Db)
99          ):{
100           sql_injection_unauthorized_access(action_form(User, Form, Db));
101         }
102
103         % Stored_XSS. We assume that (i) everything that is stored
104         % somewhere in the web application is eventually used to
105         % generate part of the content of a dynamic page, and (ii)
106         % we remove the condition that a query has been executed
107         % because we abstract from the type of storing mechanism
108         % (database, file system, etc.)
109         on( grant(?User,?Page) &
110           dishonest(?User) &
111           inputForm(?Page,?Form) &
112           storing(?Form,?Db)
113         ):{
114           stored_XSS(action_form(User, Form, Db));
115         }
116
117         % Reflected XSS
118         on( grant(?User,?Page) &
119           dishonest(?User) &
120           inputForm(?Page,?Form) &
121           printValue(?Form)
122         ):{
123           reflected_XSS(search_form(User, Form));
124         }
125
126       } % close select
127     } % close while
128   } % close body
129 }
```