Federico De Meo

A Formal and Automated Approach to Exploiting Multi-Stage Attacks of Web Applications

Ph.D. Thesis

November 5, 2018

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
Prof. Luca Viganò
Dipartimento di Informatica
Università degli Studi di Verona
Strada le Grazie 15, 37134 Verona
Italia
and
Department of Informatics
King's College London
UK

Series N°: **????**

Università degli Studi di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italia

Abstract. The complexity of modern web applications, due to the implementation of new services, has rapidly increased the need of new automatic security analysis methods and tools. Today, the leading methodology for the security analysis of web applications is a combination of vulnerability assessment and penetration testing. Vulnerability assessment has received much attention and several tools have been proposed to identify vulnerabilities. On the other hand, penetration testing has been left to the experience of the security analyst.

In this thesis, I address this problem by proposing a formal, model-based testing approach for the security analysis of web applications that can support the penetration testing phase. The approach I propose is based on the formal definition of web applications and their vulnerabilities which allow one to (i) reason about vulnerabilities of web applications and (ii) combine multiple vulnerabilities for the identification of complex, multi-stage attacks. I have developed WAFEx, an automated tool that implements my approach and I show its efficiency by applying it to real-world case studies. WAFEx was able to find previously unknown attacks, which are witness to the fact that WAFEx can generate, and exploit, attacks that, to the best of my knowledge, no other tool for the security analysis of web applications can find.

Contents

List of Figures

List of Tables

1

Introduction

In the early days of the Internet, the World Wide Web consisted only of static web pages that essentially were information repositories for sharing content with other users. The term "web site" is used to describe a collection of many web pages hosted on the same domain that could be browsed with a web browser. Today, the World Wide Web is almost unrecognizable from its early form and the term "web application" is often used instead of "web site" to describe a new type of content accessible on the web that is capable of delivering complex features compared to the ones provided by desktop software applications. The majority of web based content that we access today is delivered in form of a web application and we use web applications to deal with many different services such as bank accounts, social networks and health care. There is thus no doubt that web applications play an important role in our everyday life and have become the default technology choice adopted by many companies when deploying fast new software to the public. The ubiquity of web applications has inevitably attracted the attention of attackers who wanted to profit from the widespread of such a technology. Ensuring the security of web applications is thus an important task that must not be overlooked in the process of delivering web applications.

A number of different approaches have been proposed to test the security of web applications (*web applications*, for short), ranging from *vulnerability assessment* [83] and *penetration testing* [15], which are the two main approaches that security analysts typically undertake when assessing the security of a web application and other systems under analysis (see also [21, 35, 73, 32]), to the more formal *model-based testing* [81, 33] that has been steadily maturing into a viable alternative and/or complementary approach.

Vulnerability assessment and penetration testing do not have universally accepted definitions. I now briefly describe the two approaches in the context of this thesis and the correlation between them. A vulnerability assessment encompasses the use of automatic scanning tools to search for common vulnerabilities of the system under analysis. This approach gives a superficial overview of the security of a system and does not require many technical skills

as the automatic tools are in charge of taking care of the analysis. However, it is well known [29] that state-of-the-art automatic scanners do not detect vulnerabilities linked to the logical flaws of web applications. This means that *even if a vulnerability is found, no tool can link it to logical flaws leading to the violation of a security property.* The result of the vulnerability assessment is thus used to perform a second and more complicated step: during a penetration test (*pentest*), the security analyst defines an attack goal and manually attempts to exploit the discovered vulnerabilities to determine whether the attack goal he defined can actually be achieved. A pentest is meant to show the real damage on a specific system under analysis resulting from the exploitation of one or more vulnerabilities.

1.1 Motivation

Consider the following example, which is simple but also fundamental to understand the motivation for the analysis that I propose. Trustwave SpiderLabs found a SQL injection vulnerability in Joomla! [43], a popular Content Management System (CMS). In [79], Trustwave researchers show two things: the code vulnerable to SQL injection and how the injection could have been exploited for obtaining full administrative access. The description of the vulnerable code clearly highlights an inadequate filtering of data when executing a SQL query. The description of the damage resulting from the exploitation of the SQL injection shows that an attacker might be able to perform a session hijacking by stealing session values stored as plain-text in the database. The result of this analysis points out two problems: Joomla! is failing in (1) properly filtering data used when performing a SQL query and (2) securely storying session values. Problem (1) could have been identified by vulnerability scanners (e.g., sqlmap is able to identify the vulnerability), but *no automatic vulnerability scanner can identify problem (2) as it depends on how the web application handles sessions and thus only a manual pentesting session is effective.*

However, manual pentesting relies on the security analyst's expertise and skills, making the entire pentesting phase easily prone to errors. An analyst might underestimate the impact of a certain vulnerability leaving the entire web application exposed to attackers. *This is why it is important to develop new approaches capable of automatically identifying complex attacks to web applications.*

1.2 Contributions

My main contributions are two-fold. First, I propose a formalization to represent the behavior of the most dangerous vulnerabilities of web applications:

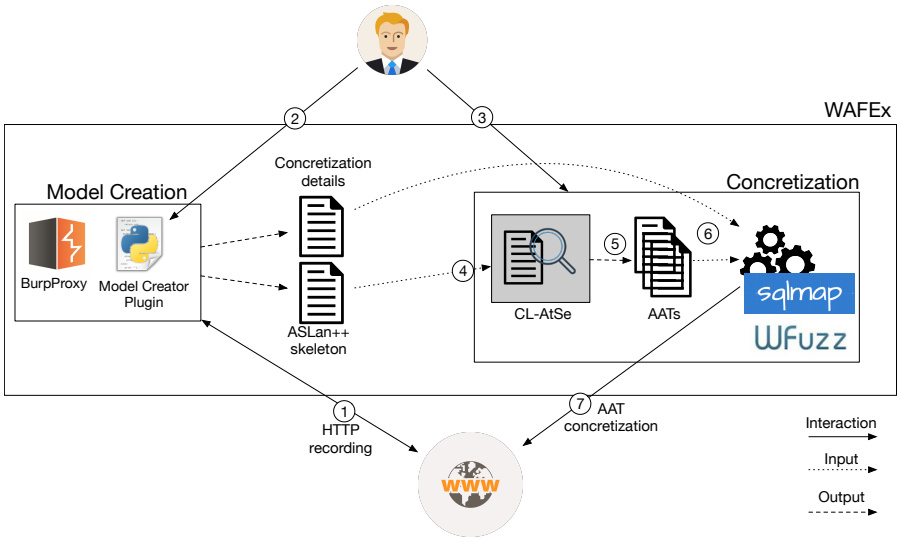


Fig. 1.1: Workflow of my approach (and of the WAFEx tool).

SQL injection (SQLi), *XSS*, *Cross-Site Request Forgery (CSRF)* and *file-system* related vulnerabilities [61]). (But note that my formalization can be fairly easily extended to consider other vulnerabilities as well.) In particular, I show how the canonical *Dolev-Yao (DY) attacker model* [27] can be used to search for multi-stage attacks where multiple vulnerabilities are combined together to violate security properties of web applications. A number of formal approaches based on the DY attacker model have been developed for the security analysis of web applications, e.g., [3, 6, 18, 24, 70, 84], but combinations of multiple vulnerabilities have never been taken into consideration by formal approaches before. It is crucial to point out that my approach does not search for payloads that can be used to exploit a particular vulnerability, but rather I propose an approach capable of automatically exploit vulnerabilities of web applications.

Second, to show that my formalization can effectively generate multi-stage attacks where multiple vulnerabilities are exploited, I have developed a prototype tool called *WAFEx (Web Application Formal Exploiter [86])*. As shown in Figure 1.1 (that I will discuss in detail in Chapter 6), WAFEx follows the canonical model-based testing approach and is capable of generating automatically different attack traces that violate the same security property. More importantly, WAFEx is capable of generating, and exploiting, complex attacks that, to the best of my knowledge, no other state-of-the-art-tool for the security analysis of web applications can find.

I tested WAFEx on well-known vulnerable web applications used to test the skills of pentesters (specifically WebGoat, DVWA and Gruyere) and WAFEx was able to identify all the vulnerabilities covered by my formalization. I then applied WAFEx to two real-world case studies: Multi-Stage (a web application that I wrote for testing) and Cittadiverona (a public web application provided by Virtuopolitan S.r.l.), and WAFEx was able to identify previously unknown attacks to Cittadiverona, which I disclosed to the provider before writing them up in this thesis.

Finally, I also contributed to the development of the *Model-based Security Testing Framework (MobSTer)*. Specifically, I worked on the definition of *actions*, a means to represent web applications, and I performed comparative tests to evaluate the effectiveness of MobSTer against state-of-the-art tools for the secure analysis of web applications.

1.3 Publications related to this thesis

Published papers:

1. Federico De Meo, Marco Rocchetto and Luca Viganò. "Formal Analysis of Vulnerabilities of Web Applications Based on SQL Injection" . In the proceedings of *Security and Trust Management (STM) (2016)* [24].
2. Federico De Meo and Luca Viganò. "A formal approach to exploiting multi-stage attacks based on file-system vulnerabilities of web applications". In the proceedings of *Engineering Secure Software and Systems (ESSoS) (2017)* [25].
3. Michele Peroli, Federico De Meo, Luca Viganò and Davide Guardini. "MobSTer: A Model-based Security Testing Framework for Web Applications" . In *Software Testing, Verification and Reliability* [66].

Submitted papers:

1. Federico De Meo and Luca Viganò. "A Formal and Automated Approach to Exploiting Multi-Stage Attacks of Web Applications". Currently under submission [26].

1.4 Synopsis

Part I — State of the art:

- Chapter 2 gives an overview of the most dangerous vulnerabilities of web applications that I formalized in my research.
- Chapter 3 gives an overview of the approaches used nowadays for analyzing the security of web applications.

Part II — Model-based Security Testing Framework (MobSTer):

- Chapter 4 describes the MobSTer framework that is used to test vulnerable entry points of web applications.

Part III — Multi-stage analysis of web applications:

- Chapter 5 introduces my formalization for exploiting multiple vulnerabilities of web applications.
- Chapter 6 describes the details of my tool WAFEx and the experimental results of applying WAFEx to some real-world case studies.
- Chapter 7 discusses related works.
- Chapter 8 summarizes the contributions of this thesis.
- Chapter 9 discusses some possible future work.

Appendix A:

- Appendix A describes the formal model of three case studies (DVWA, WebGoat and Gruyere) that I used to test the implementation of WAFEx.

Part I

State of the art

2

Web applications security

This chapter gives an overview of the main security issues relevant to the security of web applications with particular emphasis to some classes of vulnerabilities which I closely studied and that characterize my research. I do not give a specific order to the vulnerabilities that I present but I rather collect them in three categories representing three components of web applications: database , file-system and client. This categorization aims at giving a coherent and uniform starting point to reason about vulnerabilities of web applications that will also be used later on in Chapter 5 where I present the main contribution of this thesis For each category, I describe relevant vulnerabilities that aim at compromise their security. More specifically, I consider two security properties of interest for the collection of the vulnerabilities:

- *Authentication bypass*: the attacker has access to a restricted area without providing valid credentials.
- *Confidentiality breach*: the attacker has access to content stored in the web application's file-system that is not meant to be publicly available.

Along with relevant vulnerabilities, I describe mitigation techniques to ensure a safe interaction with each component. Finally, the vulnerabilities collected in the this chapter can also be found in well known classification such as the OWASP Top Ten project [61] which aims at collecting the ten most critical vulnerabilities of web applications by providing a description, examples and guidance on how to avoid them.

2.1 Database related vulnerabilities

Databases are employed by almost every web application to store and organize information delivered to users. The most used language employed by web applications to query the content of a database is Structured Query Language (SQL). SQL is an interpreted language that can be used to perform a variety

of operations on a database such as read, insert, update and delete. Developers commonly implement the interaction with a database by writing SQL statements as concatenation of SQL commands and user-supplied data. However, user-supplied data, if used maliciously, might contain additional SQL commands that will change the behavior of the resulting SQL statement leading to a SQLi vulnerability.

SQLi is one of the most widespread security vulnerabilities of the web: according to the Open Web Application Security Project (OWASP), SQLi is the most critical threat for the security of web applications [61], and the MITRE corporation lists improper SQLi neutralization as the most dangerous mistake in web application's development [22]. The first definition of SQLi dates back to 1998, when Jeff Forristal wrote about this technique in the Phrack ezine [34]. After 20 years, also due to the increasing complexity of modern web applications, SQLi is still widely exploited and can be very difficult to detect. A variety of SQLi scanners, such as sqlmap [74] and sqlninja [75], have been proposed in recent years to search for SQLi injection points and payloads and a number of general classifications based on the payloads of the SQLi (and the scenario in which they are exploited) have been put forth, e.g., in [38, 61]. In this section, I describe the five main different categories of SQLi techniques needed to understand my formalization: (i) Boolean-Based Blind, (ii) Time-Based, (iii) Error-Based, (iv) Union Query, (v) Second-Order and (vi) Stacked Queries.

2.1.1 Boolean-based blind SQLi

In this type of SQLi, an attacker inserts into an HTTP parameter, which is used by a web application to write a SQL query, one or more valid SQL statements that make the `WHERE` clause of the query evaluate to true or false.[1] By interacting multiple times with the web application and comparing the responses, the attacker can understand whether or not the injection was successful.

As an example, consider a web application that shows a page *login.php* for performing the login. Once the user submits hes credentials (username and password), the web application searches in the database a tuple that matches the given username and password by using the query `SELECT * FROM users WHERE users.username='username' AND user.password='password'`. Assume that the web application replies with page `login.php` if no tuples are returned by the database and with another page, `dashboard.php`, if a tuple is returned. Hence, if the attacker injects a payload such as `' OR users.username='admin'` in one of the fields of the form and is redirected to *dashboard.php*, then he will know that `admin` is a valid value in the database for column `username`.

[1] Since all the state-of-the-art databases (SQL Server, SQLlite, MySQL, PostgresSQL and Oracle) are vulnerable to SQLi, I don't distinguish between different SQL dialects and simply write "SQL query".

2.1.2 Time-Based SQLi

This injection is quite similar to Boolean-Based blind but does not need the web application to have a Boolean behavior. This injection technique is usually exploited in scenarios where there is no (trivial) way for the attacker to understand whether, after the SQLi, the database has produced any tuples as a response of a query. To overcome this lack of information from the web application, the attacker appends a timing function to the validity value of a Boolean clause. In this way, the attacker tricks the database into waiting, after the submission of the query by the web application, for a predefined amount of time if there is a tuple as a response to the query. The attacker can then infer whether the value of the query was true (i.e., observing a delay in the response) or false (i.e., no delay is observed).

As an example, consider a web application that replies with a page `search.php` independently on whether a tuple is returned by the database. Whatever the input provided by an attacker, he will always receive the same page. The attacker might now try to use a temporal operator provided by SQL in order to distinguish the behavior of the query. Hence, the attacker might inject a payload such as `OR IF(username=admin)WAIT 60s` and, if `admin` is a valid value in the database column `username`, the attacker will observe a delay of around 60 seconds before getting any answer from the web application. The attacker can then infer that `admin` is actually a valid value.

In real case scenarios, a Boolean-Based blind is preferred whenever possible because it is faster than a Time-based.

2.1.3 Error-Based SQLi

During the development and testing phases of a web application, it can be useful for developers to be able to inspect errors returned by the database within the web application itself. This common practice, while very helpful, should be limited to a testing environment and the error pages should be removed, or errors should be limited to standard error messages, once the web application moves to deploy. When error pages are not removed and, instead, exposed to the Internet, some error messages provided by the database could be exposed, thus giving to an attacker the possibility of exploiting *Error-based SQLi*. In this type of injection, the attacker tricks a database into performing operations that result in an error and then he gains information from the error messages produced by the database itself. The attacker induces the generation of an error that contains some pieces of information stored in the database and, with subsequent interrogations, he will eventually gain all the data stored in the database.

As an example, consider an attacker who wants to find out the first username in the table `usernames`. The attacker might inject, in a login form, a payload that tricks the web application into evaluating the query `SELECT * FROM (SELECT username FROM usernames LIMIT 1)`, which generates the

error `Error: table adminUsername unknown` because an invalid table is selected (resulting from the inner query `SELECT username FROM usernames LIMIT 1`), where `adminUsername` is the first username found in the table usernames.

2.1.4 UNION Query-Based SQLi

This is a particular type of SQLi where the attacker injects a SQL `UNION` operator to join the original query with a malicious one. The `UNION` operator only applies to `SELECT` queries and is used by the attacker to override the result of the original query. Since the attack overwrites the result of the original query, this kind of injection requires the web application to print the result of the query within the returned HTML page. This behavior allows the attacker to actually gain information from the database by reading them within the web application itself.

As an example, consider the query `SELECT nickname FROM users WHERE id=?`. An attacker can inject `1 UNION ALL SELECT creditCardNumber FROM CreditCardTable` as the value of `id` which forces the query to return the credit card numbers.

2.1.5 Second-Order SQLi

This injection doesn't have a direct effect when submitted but is exploited in a second stage of the attack. More specifically, in some cases a web application may correctly handle and store a SQL statement whose value depends on user input. Afterwards, another part of the web application that doesn't implement a control against SQLi might use the previously stored SQL statement to execute a different query and thus expose the web application to a SQLi. This attack is quite complex and requires the attacker to have a deep knowledge of how data are stored and used in the web application. Automated web application security scanners generally fail to detect this type of SQLi (e.g., [75] does not support second-order) or may need to be manually instructed to check for evidence that an injection has been attempted (e.g. [74] provides the option `--second-order` used to specify the resulting URL page to search for second-order attempt).

As an example, consider a web application that allows users to register new accounts and lets assume that a user with username `admin` is already registered in the web application and that the registration page implements proper countermeasures against SQLi. An attacker can then register a new user with username `admin'#` (where `#` is the comment delimiter character). The web application will then store in the database a new entry where the value for the username would be `admin'#`. At this point, the attacker can log in as `admin'#` and change his password, meaning changing the password of the user admin'#. If the query that handles the update is unsafe, meaning

does not validate its values, it might look something like `UPDATE users SET password='newpassword' WHERE username='admin'#'`. The database will interpret the `WHERE` clause as `username='admin'` (because everything after the `#` character is considered a comment), so that the attacker is actually changing the password of the user `admin` and not of user `admin'#`.

2.1.6 Stacked Queries SQLi

This injection allows the attacker, whenever he finds an injection point, to execute an arbitrary query completely different from the original one. In SQL, the semicolon character `;` is used to mark the end of an SQL statement and the beginning of a new one. An attacker might exploit the semicolon character and force the execution of a completely new query. This type of SQLi is probably the most powerful one as, when applicable, it allows the attacker to completely change the injected query. However, it is important to highlight some limitation of when Stacked-queries can be executed. In fact, most of the time it is not possible to exploit this SQLi technique since the database or the framework used to interact with the database might not support the execution of multiple queries.

As an example, consider the query `SELECT title, description FROM articles WHERE id=id`. An attacker might inject the statement `1;DROP TABLE articles` causing the execution of a drop table query which erases the entire table `articles`.

2.1.7 Prevention techniques

Despite all the different techniques that can be used to exploit a SQLi entry point, avoiding SQLi attacks is theoretically quite straightforward. In fact, to avoid these attacks, developers can use *sanitization functions* or *prepared statements*. The general idea behind the prevention of SQLi attack is to not evaluate the injected string as a SQL command.

A sanitization function takes the input provided by the user and modifies it in such a way to remove the threat posed by those characters that can be used to alter a SQL statement. Whenever one of these characters is detected in user-supplied input, a sanitization function alters the input by either encoding the character, so to avoid it being interpreted as a SQL command, or deleting it. Sanitization functions are not the best option when dealing with SQLi when compared with *Prepared statement*. This because the application of such a function might be useless when the function is not properly implemented or does not consider some cases. However, they are an applicable option when prepared statements are not provided by the language.

Prepared statements (also known as *parameterized statement*), on the other hand, are the best option for preventing a SQL statement to be exposed to SQLi. They are mainly used to execute the same query repeatedly maintaining efficiency over time but, because of the inner execution principle

of prepared statements, they are immune to SQLi attacks. The execution of a prepared statement consists mainly in two steps: *preparation* and *execution*. In the preparation step, the query is evaluated and compiled, waiting for the parameters for the instantiation. During the execution step, the parameters are submitted to the query and handled as data and thus they cannot be interpreted as SQL commands anymore. However, it is important to point out that also prepared statements might be susceptible to SQLi if wrongly used within the web application. Consider the PHP code in Listing 2.1 showing a simple example of a wrongly implemented usage of prepared statement. The variable $pdo is initialized to open a connection with the database (line 1). Variables $setStr and `$id` are initialized by taking value from user supplied GET queries (lines 2-3). Finally, a prepared statement is executed where the variable $setStr is concatenated to the SQL statement, thus used incorrectly as it can be used to inject SQL commands, while the variable $id is correctly used as a parameter of the prepared statement (line 4).

Listing 2.1: PHP code showing a wrong use of prepared statements

```
1 $pdo = new PDO('mysql:dbname=test;host=localhost', '
    root', '');
2 $setStr = $_GET['name'];
3 $id = $_GET['id'];
4 $pdo->prepare("UPDATE users SET name='$setStr' WHERE
    id = :id")->execute($id);
```

2.2 File-system related vulnerabilities

Modern web applications make intensive use of functionalities for reading and writing content from the *web application's file-system* (i.e., the file-system of the web server that hosts the web application). *Reading* from and *writing* to the file-system are routine operations that web applications perform for many different tasks. For instance, the possibility of dynamically loading resources based on runtime needs is commonly adopted by developers to structure the web application's source code for better and stronger re-usability. Similarly, for what concerns writing, an increasing number of web applications allow users to upload (write) content that can be shared with other users or can be available from a web browser as in a cloud service. Reading and writing functionalities are offered by most server-side programming languages for developing web applications such as PHP [67], JSP [57] or ASP [48]. Modern database APIs also provide a convenient way to interact with the file-system (e.g., backup or restore functionalities), but such APIs also increase the attack surface an attacker could exploit. Whenever an attacker finds a way to exploit vulnerabilities that allow him to gain access to the web application's file-system, the security of the whole web application is put at high risk. Indeed, both OWASP [61] and MITRE [22] list vulnerabilities that compromise

the file-system among the most common and dangerous vulnerabilities that afflict the security of modern software.[2]

Some classifications about file-system related vulnerabilities have been given in the past [60, 59], but classifying vulnerabilities based on the security property that is violated (e.g., authentication) rather than the functionality being exploited (e.g., file-system functionality). I have identified five main categories of interest, which are described below.

2.2.1 Directory Traversal (a.k.a. Path Traversal)

Reading and writing operations performed by a web application are intended to occur in a restricted directory where the web application actually resides. This location is referred to as the *root directory* of the web application. A Directory Traversal vulnerability refers to a lack of authorization controls when an attacker attempts to access a location that is intended to identify a file or directory stored in a restricted area outside of the web application's root directory. Whenever the access permissions of a web application are not restricted in such a way that they only allow authorized users to access specific files, an attacker might be able to craft a payload that allows him to access restricted files located outside the web application's root directory. Directory Traversal payloads make use of special characters such as the double dots "`..`" and the forward slash "`/`" separator, which, when combined, allow the attacker to specify arbitrary locations that can escape outside the root directory of a web application. Directory Traversal attacks can be further divided in *Relative* and *Absolute*, depending on whether the payload refers to a relative or an absolute path. Since a Directory Traversal vulnerability refers to a lack of authorization permissions, to actually exploit it, it is necessary for an attacker to find an entry point that allows him to send input to the web application that is then used to create a file-location string. This means that a Directory Traversal vulnerability is always exploited in combination with another vulnerability that provides such an entry point to the attacker. For example, imagine that `index.php?load=file` refers to a web page `index.php` that dynamically loads the file specified by the value of `load`. An attacker might modify this value and use it as input vector to exploit a Directory Traversal vulnerability.

2.2.2 SQLi

Web applications make use of databases in order to store data. This allows for functionalities such as blog posting, forum discussions, etc. As described

[2] The Top 10 compiled by OWASP is a general classification and it does not include a specific category named "file-system vulnerability"; however, "Injections", "Broken Authentication and session Management", "Security misconfiguration" (just to name a few) can all lead to a vulnerability related to the file-system.

in §2.1, querying a database is performed using the SQL language and whenever a query is created using user-supplied data, SQLi attacks could be possible. Most modern databases provide APIs that extend the expressiveness of SQL by allowing SQL code to access a web application's file-system for reading and writing purposes. Reading APIs allow developers to produce code that can retrieve content stored in the web application's file-system and load it in the database. This is particularly convenient when a web application needs to load bulks of data into the database, e.g., as part of an initialization or restoring process. Writing APIs allow developers to produce code that can save content from the database to the web application's file-system. This is also particularly convenient for features such as backup or upload functionalities. When an attacker finds an SQLi entry point, he can inject arbitrary SQL syntax that modifies the behavior of the original query. Attackers mainly exploit SQLi to bypass authentication mechanisms or to extract data from the database, but, as there is no limit on the SQL syntax that could be injected, it might also be possible to exploit reading and writing APIs to access the underlying file-system [23].

As an example, consider the MySQL database [55], which has the built-in API `LOAD_FILE()` for reading text or binary files from the file-system [56]. In the case of a `UNION` query-based SQLi, an attacker might inject the payload `1 UNION ALL SELECT 1,LOAD_FILE('/etc/passwd'),3,4 FROM mysql.user--` that will give him reading access to the file `/etc/passwd`. Similarly, MySQL provides APIs for writing to the file-system with the `SELECT ... INTO` statement, which enables the result of a `SELECT` query to be written to a file.

2.2.3 File Inclusion

All programing languages for the development of web applications support functionalities for structuring code into separate files so that the same code can be reused at runtime by dynamically including files whenever required. A File Inclusion vulnerability refers to a lack of proper sanitization of user-supplied data during the creation of a file location string that will be used to locate a resource meant to be included in a web page. When the file location depends on user-supplied data, an attacker can exploit it and force the inclusion of files different from the ones intended by the developers. File Inclusion might allow an attacker to access arbitrary resources stored on the file-system and to execute code. File inclusion attacks can be further divided in *Local file inclusion* and *Remote file inclusion*, which are respectively used to force the inclusion of files stored in the local server or to include file stored in a remote location.

As an example, consider the PHP code in Listing 2.2. Line 1 gets a user-supplied parameter $_GET['user'] and stores it into the variable $username that is then used to create a file location in line 2, which is in turn used to include a resource in the current page in line 3. If an attacker injects the

payload .htaccess as the value for the parameter user, he might be able to access the file `.htaccess`, which contains configuration specific for the current directory [4].

Listing 2.2: PHP code vulnerable to file inclusion

```
1 $username = $_GET['user'];
2 $filepathname = "/var/www/html/".$username;
3 include $filepathname;
```

File inclusion can be combined with Directory Traversal vulnerability, allowing an attacker to gain access to resources stored on the web application's file-system but outside the web application's root folder. Consider again Listing 2.2 and suppose that the web server hosting the web application is a unix server. The attacker might then inject a malicious payload such as `../../../../etc/passwd`, where `/etc/passwd` is the common location pointing to a text-based database listing the users of the system. Assuming that the root directory of the web application is located at `/var/www/html/site/`, the PHP code will try to include the file `/var/www/html/site/../../../../etc/passwd`, and the path is translated into `/etc/passwd`, forcing the web application to include the file and thus giving the attacker access to the list of users for the web server.

2.2.4 Forced Browsing (a.k.a. Direct Request)

This vulnerability refers to a lack of authorization controls when a resource is directly accessed via URLs. This lack of authorization might allow an attacker to enumerate and access resources that are not referenced by the web application (thus not directly displayed to the users through the web application) or that are intended to be accessed only as a result of previous HTTP requests. By making an appropriate HTTP request, an attacker could access resources with a direct request rather than by following the intended path. The lack of authorization controls comes from the erroneous assumption that resources can only be reached through a given navigation path. This wrong assumption leads developers to implement authorization mechanisms only at certain points along the way for accessing a resource, leaving no controls when a resource is directly accessed.

As an example, consider a web application where the URL `http://vuln.com/admin/index.php` points to the login page and `http://vuln.com/admin/admin.php` points to the administration page accessible once the login has succeeded. When Forced Browsing is possible, an attacker might be able to directly access the administration page by requesting the URL `http://vuln.com/admin/admin.php` without first logging in. If the `admin.php` page does not verify whether the request is made by an authorized user, the attacker has skipped the login process provided by `http://vuln.com/admin/index.php`.

Another example is a development environment that is supposed to be accessible only by developers. Developers usually erroneously assume

that since the development environment is not directly accessible from the main website, users have no means to access this area. However, an attacker might try to guess the name of the development environment (e.g., `http://vuln.com/dev/`) and increase the chance of having unauthorized access to the web application.

2.2.5 Unrestricted File Upload

A widespread feature provided by web applications is the possibility of uploading files that will be stored on the web application's file-system. An Unrestricted File Upload vulnerability refers to a lack of proper file sanitization when a web application allows for uploading files. The consequences can vary, ranging from complete takeover with remote arbitrary code execution to simple defacement, where the attacker is able to modify the content shown to users by the web application.

As an example, consider a web application that allows users to upload images (e.g., an avatar to customize the user's profile). Listing 2.3 gives the HTML code of the file upload form while Listing 2.4 gives the PHP code that performs the upload. With refer to Listing 2.4, lines 1-2 define the location where the file will be uploaded and line 3 performs the actual upload. However, line 3 does not perform any control over the type of file being uploaded, paving the way to Unrestricted File Upload Attacks.

Listing 2.3: HTML code that shows a file upload form

```
1 <form enctype="multipart/form-data" action="uploader
    .php" method="POST">
2 Choose a file to upload: <input name="uploadedfile"
    type="file" /><br />
3 <input type="submit" value="Upload File" /> </form>
```

Listing 2.4: PHP code for uploading a file

```
1 $path="uploads/";
2 $target_path = $target_path.basename($_FILES['
    uploadedfile']['name']);
3 if(move_uploaded_file($_FILES['uploadedfile']['
    tmp_name'],$target_path)){
4  echo "file uploaded!";
5 }else {
6  echo "error uploading file!";
7 }
```

Unrestricted File Upload can also be used in combination with Directory Traversal, allowing the attacker to overwrite arbitrary files stored in the web server. However, modern web application programming languages (like PHP) perform a sanitization on the uploaded file path string by removing any special

characters such as "`..`" and "`/`" used to change the current path, making the exploitation of a Directory Traversal less likely to happen.

2.2.6 Prevention techniques

The most effective solution to avoid undesired access to the file-system is to not use user-submitted input directly in reading and writing operations. This might not always be possible, or convenient, so to restrict access to the file-system it is suggested to implement a mapping between identifiers and files, and use identifiers when performing requests. Such a mapping, when properly implemented, ensures that users cannot submit an arbitrary file name and have access to it. Moreover, it is always important to implement the right credentials to all the files and folders within the scope of the web application. When the server of the web application is running, the user running the server should be restricted in accessing only the files required to properly run the web application.

2.3 Client-side related vulnerabilities

Users interact with a web application by means of a web browser and thus an important aspects that needs to be taken into consideration when dealing with the security of web applications is the interaction with the web browser. In this section, I describe two aspects of web applications that increase the attack surface an attacker can exploit: *HTTP cookies* and *client-side scripting*.

The HTTP protocol was developed as a stateless protocol for transferring text messages across a network of computers. The general idea behind it is pretty simple, an HTTP request is made by a client and an HTTP response is generated by a server to answer the request. Each pair of request-response is an independent transaction that is not related to any previously made request-response. The independencebetween one pair of request-response and another does not allow a server to know if the same client is performing consecutive requests, meaning that it is not possible to deliver an user experience where, for example, an user *Bob* performs one HTTP request and then another HTTP request and the web application remembers that *Bob* performed two HTTP requests. HTTP Cookies were designed to circumvent the stateless nature of HTTP and deliver a more stateless experience. An HTTP Cookie is a piece of information that is transmitted from the web application to the web browser and used to implement the concept of *session*. A session allows a web application to remember when a certain user performed a certain request. When a request is made to a web page that creates a session, the web page allocates a memory area and assigns to that area an identifier. The identifier is then sent back to the client as an HTTP Cookie within the response for that request. When the browser receives an HTTP Cookie, it stores it in its internal memory and associate that Cookie with the domain that generated

it. Whenever the browser generates a new request to a domain for which a Cookie has been saved, the web browser automatically sends the Cookie back to the web application. The web application receives the Cookie containing the session identifier and uses it to retrieve the information stored within the associated memory area thus implementing a stateful browsing experience.

To further improve the way users interact with web applications, new technologies where proposed for creating a more appealing experience. These new technologies take advantages of the user's browsers and allow developers to delivery content such as floating menus, draggable windows, fade in or fade out images with the intent of creating more friendly and beautiful user interfaces. Technologies such as JavaScript [52], ActiveX [47], Silverlight [49], Flash [2], to name a few, were proposed, however only one survived and is today the most relevant and widely adopted. JavaScript is a client-side interpreted programming language capable of integrating with HTML code and providing great interactive content to users.

Both HTTP Cookies and client-side scripting languages opened the path to attacks that could be carried out on web applications. There is, however, an important difference to highlight with the vulnerabilities I presented so far and the one I discuss in this section. The attacks described in §2.1 and in §2.2 are performed by an attacker who is directly targeting the web application by exploiting a false sense of trust from the web application on the data received by users. The vulnerabilities I describe in this section, on the other hand, involve the interaction with clients, where the attacker exploits a false sense of trust from the clients on the data received from a web application.

2.3.1 Cross-Site Scripting (XSS)

XSS vulnerabilities are a type of injection vulnerabilities that force a web application into displaying malicious code that is then sent and executed by web browsers; thus, the web application hosts the payload while the main target is the web browser. For an attack to be successful, a user has to visit an infected page containing malicious JavaScript, which is then executed by the web browser of the user.[3] In the past, XSS was considered a not serious vulnerabilities since it does not directly target the web application ([76, p. 433]). However, XSS can be quite dangerous and both the OWASP Top 10 and [61] and MITRE [22] list XSS as a common and high risk vulnerability. In fact, when an XSS vulnerability can be exploited, for example, it can be used to get access to information stored on the client's web browser. As already stated, browsers store information such as session cookies that can be retrieved by an XSS attack to perform a so called *Session Hijacking* attack. An attacker that steals the session cookie of an user, can interact with the web application pretending to be that user. However, XSS can be used to perform many

[3] JavaScript is the most used language for exploiting XSS since it is widely supported by all modern web browsers, but any supported client-side language could be used.

other nefarious actions that sometimes go overlooked. Some of the common malicious actions that can be carried out by an XSS attack includes, but are not limited to:

- redirecting the navigation of a user by forcing the browser into loading a different page;
- stealing the history of an user that can be used to conduct phishing attacks;
- getting information on the local intranet of a user;
- performing post scanning.

There are different scenarios that lead a web application to become the unintended host of malicious client-side code that can be generalized in the two main categories under which XSS vulnerabilities are divided: *Reflected XSS* and *Stored XSS*.

A *Reflected XSS* happens when a web application implements a dynamic page that takes user-supplied input and simply displays it in a web page. User supplied input might contain malicious client-side code that is thus executed by the browser. This behavior can be found in many different type of web pages. Consider for example the URL `http://127.0.0.1/mysearch.php?search=hello+world` where a request to a search engine is performed looking of the text `hello+world`. The result displayed by the browser might look like the one shown in Figure 2.1 where the text `hello+world` was displayed back in the response web page. An attacker might exploit this behavior and insert a valid HTML tag enclosing JavaScript code causing the execution of arbitrary client-side code. Consider a similar request where `hello+world` is replaced by `<script>alert('xss');</script>`. The web page displaying the result of the search will display the text being searched back to the user causing the browser to execute the JavaScript code as shown in Figure 2.2.

To better understand reflected XSS and the difference with the vulnerabilities discussed so far, Figure 2.3 shows the anatomy of a reflected XSS attack. A reflected XSS vulnerability can be represented in in five main steps:

1. The user logs in the web application;
2. The attacker sends a crafted URL containing malicious JavaScript to the user who logged in. This steps involve a communication between the attacker and the user which include, but is not limited to, phishing email;
3. The user performs a request to the URL received from the attacker;
4. The web application processes the request and responds to the user by reflecting the malicious JavaScript in the response;
5. The user receives the response from the web application which causes the browser to execute the embedded JavaScript;
6. Finally, the JavaScript code performs a malicious actions in the user's browser.

In a stored XSS attack, the attacker sends a request containing malicious client-side code to a web application, the web application stores (typically in

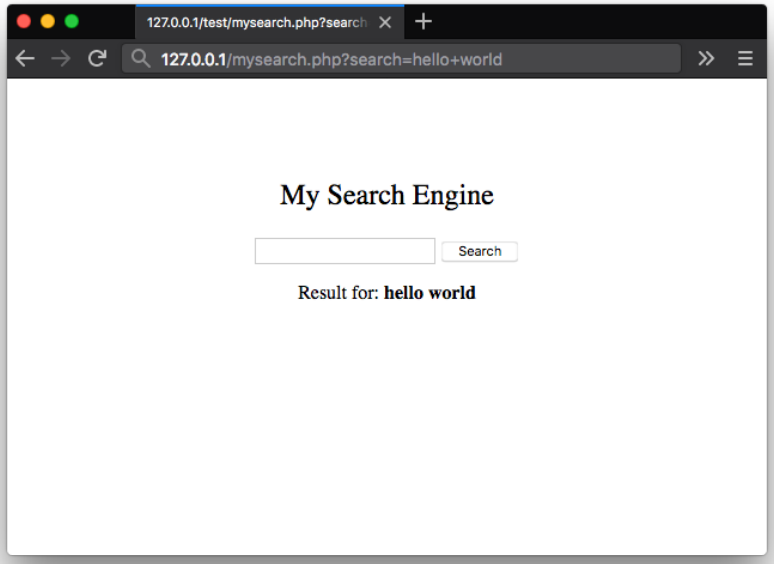


Fig. 2.1: Example of a search engine displaying back the value of a search text

the database) the value received in the request, and then, when other users are browsing the web application, the malicious content is retrieved and delivered to users. Consider for example a forum that allows users to communicate with each other. Forums are generally divided in threads where users can create new topics. When a new topic is created by a user, the web application stores in the database the messages submitted by the users so to make them available to other users. If the web application is vulnerable to a stored XSS attack, it might be possible for an attacker to submit malicious client-side code that will be stored within a topic and then executed by every user browsing that topic. A key difference distinguish between reflected and stored XSS: in the former, the attacker has to find a way to force the user into performing a request to a crafted URL while in latter, it is required to force the web application into accepting a malicious input that will be displayed as a response to request performed by another user. This make stored XSS potentially more deadly as it might spread faster and target more users. The anatomy of a stored XSS can be summarized with seven main steps as shown in Figure 2.4:

1. The attacker sends a request to the web application containing a malicious JavaScript code;

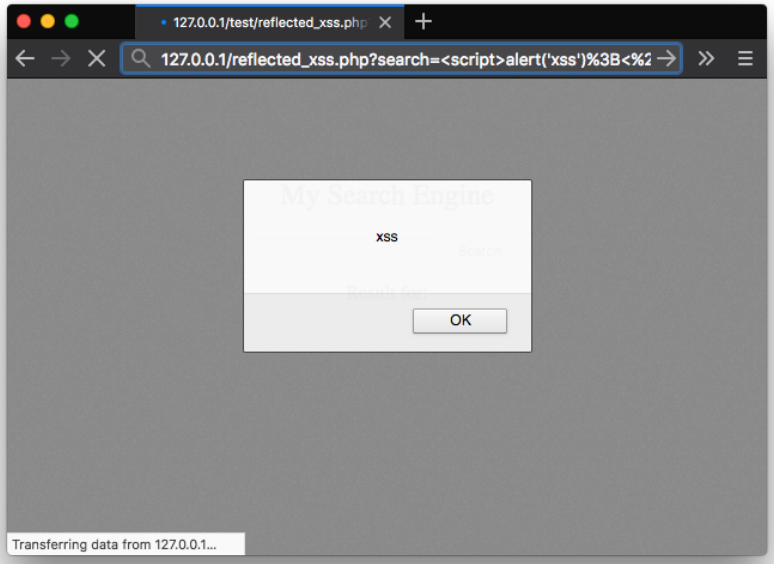


Fig. 2.2: Example of a search engine vulnerable to reflected XSS displaying a JavaScript alert

2. The web application receives the request coming from the attacker and stores (typically in the database) the malicious JavaScript code;
3. The user starts a communication with the web application and logs in it;
4. The user browses the web application and eventually accesses the content that was stored by the attacker;
5. The web application answers to the user by displaying the malicious JavaScript code within an HTML page:
6. The user receives the response from the web application which causes the browser to execute the JavaScript code embedded in the HTML page;
7. Finally, the JavaScript code performs a malicious actions in the user's browser.

As a final note, it is worthmentioning a third attack technique used to exploit JavaScript code that, thanks to its increased popularity, has gained a category of its own. *DOM Based XSS* is an attack where the payload is executed in the Document Object Model (DOM) and not in the HTML code. In this particular type of XSS, the payload will not be found in the response to an HTTP request but can only be observed at runtime or by investigating the DOM.

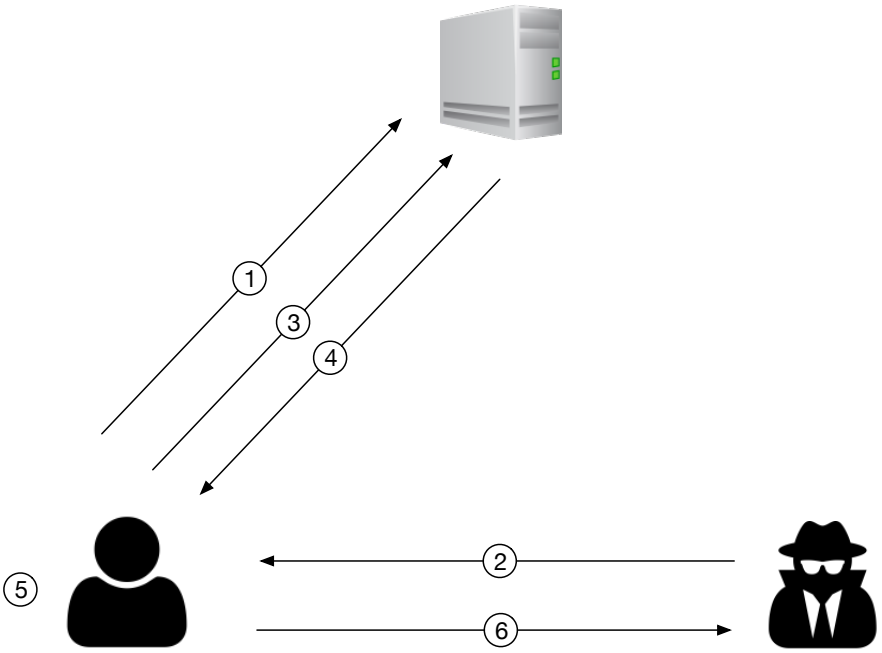


Fig. 2.3: Anatomy of a Reflected XSS attack

Consider a request to page `http://127.0.0.1/xss.html` containing the HTML code in Listing 2.5. Line 4 takes the value of document.baseURI and writes it in the DOM of the web page. If an attacker sends a request like `http://127.0.0.1/xss.html#<script>alert('xss');</script>`, the JavaScript line 4 in Listing 2.5 will get executed resulting in the string `<script>alert('xss');</script>` being written in the DOM of the web page which in turns will show a pop-up window like the on in Figure 2.2.

Listing 2.5: Example of a code vulnerable to DOM Based XSS

```
1 <html>
2   <head>
3     <script>
4       document.write("<b>URL</b>:" + document.baseURI);
5     </script>
6   </head>
7 </html>
```

is executed in the web browser as the result of modifying the Document Object Model (DOM).

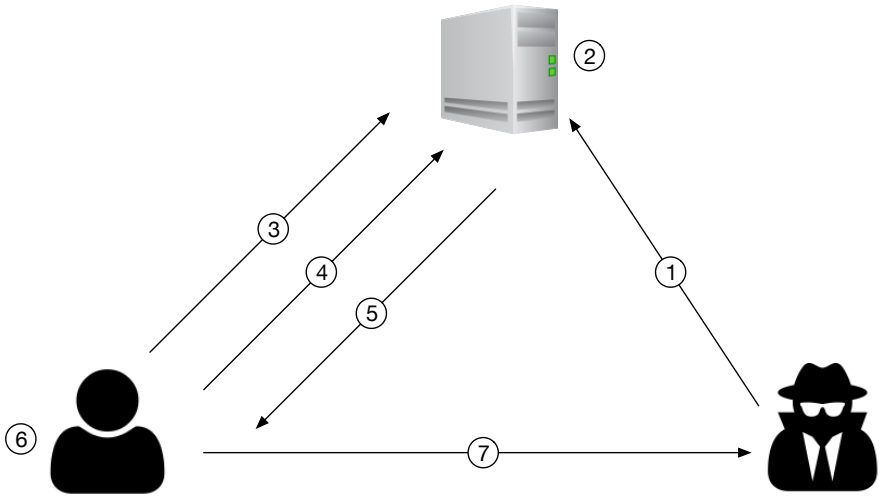


Fig. 2.4: Anatomy of a stored XSS attack

2.3.2 Cross-Site Request Forgery (CSRF)

In CSRF the attacker takes advantage of another user's session by forcing that user into performing arbitrary HTTP requests to the web application. Recall that web applications run over the HTTP protocol that, by its own nature, is a stateless protocol. HTTP Cookies have been implemented to circumvent the stateless nature of HTTP and allow developers to create web applications capable of remembering when a user performs an action (such as logging in). HTTP Cookies are automatically managed by web browsers and whenever a new request is performed to a web application that generated a cookie, the browser sends the cookie along with the request. A CSRF attack exploits the trust of a web application believing that the request it received was indeed willingly generated by the user who performed the request.

The exploitation of a CSRF attack can be very similar to the exploitation of a reflected XSS where the attacker sends a crafted URL to the user. In general, however, it can be very different. Consider a web application that implements an administrative area where only authorized users can create new users. The HTTP request for creating a new user is shown in Listing 2.6. The HTTP request is a `POST` request to the page `/admin/createUser.php` (line 1) and contains an HTTP Cookie identified by `PHPSESSID` (line 3) and the username and password of the user to create (line 8). It is important to notice the value of the HTTP header `Referer` which shows the page that generated the request and, in this case, the request was generated by the page located at `http://vuln.com/admin/createUser.php` (line 3). The credentials `bob` and `bobpasswd` are user-supplied input while the value of

the HTTP Cookie `8299BE6B260193DA076383A2385B07B9` is automatically included in the request by the web browser (line 4).

Listing 2.6: Example of an HTTP request for creating a new user

```
POST /admin/createUser.php HTTP/1.1
Host: vuln.com
Referer: http://vuln.com/admin/createUser.php
Cookie: PHPSESSID=8299BE6B260193DA076383A2385B07B9
Content-Type: application/x-www-form-urlencoded
Content-Length: 83

username=bob&password=bobpasswd
```

Imagine now that the attacker creates a malicious web page capable of forging a similar HTTP request and stores it at `http://evil.com/csrf.html`. Consider the HTML code in Listing 2.7 where there is a form pointing to `http://vuln.com/admin/createUser.php` (line 3) with two hidden field named `username` with value `charlie` (line 4) and `password` with value `letmein` (line 5). Finally, a piece of JavaScript automatically submit the form without need of user interaction (line 8). If a user, who previously logged in to `http://vuln.com` with administrative credentials, loads the malicious page located at `http://evil.com/csrf.html`, will be forced in generating the HTTP request shown in Listing 2.8. The request is performed to the page `/admin/createUser.php` (line 1) using the same value for the HTTP Cookie that was used in the previous HTTP request (line 4). The credentials sent within the request are, for the username `charlie` and for the password `letmein` (line 8). An important difference between the request in Listing 2.6 and the on in Listing 2.8 is the value of the header field `Referer` which, in this new request, assumes the value `http://evil.com/csrf.html` (line 3).[4]

Listing 2.7: Example of an HTML code capable of forging a POST request

```
1  <html>
2   <body>
3    <form action="http://vuln.com/admin/createUser.php
         " method="POST">
4     <input type="hidden" name="username" value="
          charlie">
5      <input type="hidden" name="password" value="
           letmein">
6    </form>
7    <script>
8      document.forms[0].submit();
9    </script>
10  </body>
```

[4] Note that the HTTP header `Referer` can, in genera, be forged and should not be considered trusted for preventing CSRF.

```
11 </html>
```

Listing 2.8: Example of a CSRF HTTP request for creating a new user

```
POST /admin/createUser.php HTTP/1.1
Host: vuln.com
Referer: http://evil.com/csrf.html
Cookie: PHPSESSID=8299BE6B260193DA076383A2385B07B9
Content-Type: application/x-www-form-urlencoded
Content-Length: 83

username=charlie&password=lermein
```

The overview of the anatomy of a CSRF attack is shown in Figure 2.5:

1. The attacker creates a malicious HTML page capable of forging an HTTP request (like the one in Listing 2.7);
2. The user logs in the web application using legitimate credentials;
3. The user, during his browsing on the internet, ends up loading the malicious page created by the attacker. This step can be achieved, similarly to reflected XSS, with an interaction between attacker and client which include, but is not limited to, phishing email;
4. The malicious HTML page is then sent back to the user;
5. Finally, the malicious HTML page causes the forging of a request towards the web application the user is currently logged in.

2.3.3 Prevention techniques

The most effective defense against XSS attacks is to never display user-supplied data without proper validation and sanitization. To eliminate both reflected and stored XSS vulnerabilities, developers have to identify all inputs that is then copied into a response and sent back to the user and implement proper data validation.In particular, the web application should HTML-encoding problematic characters when sending user-supplied data. HTML-encoding ensures that browsers handle potentially malicious characters in a safe way, treating them as part of the content of the HTML document and not as part of its structure. Table 2.1 shows the main problematic characters with the corresponding HTML-encoding.

On CSRF attacks, the problem arises because browsers automatically submit HTTP Cookies back to the web application that issued them with a subsequent request. The most effective defense against CSRF attacks is to include an additional piece of information to supplement HTTP Cookies. In particular, random tokens are generated and transmitted to users as hidden fields in the response of an HTTP request. When a new request is submitted, the browser has to send the random token together with the HTTP Cookie. The web application, in addition to validating the session by checking the HTTP

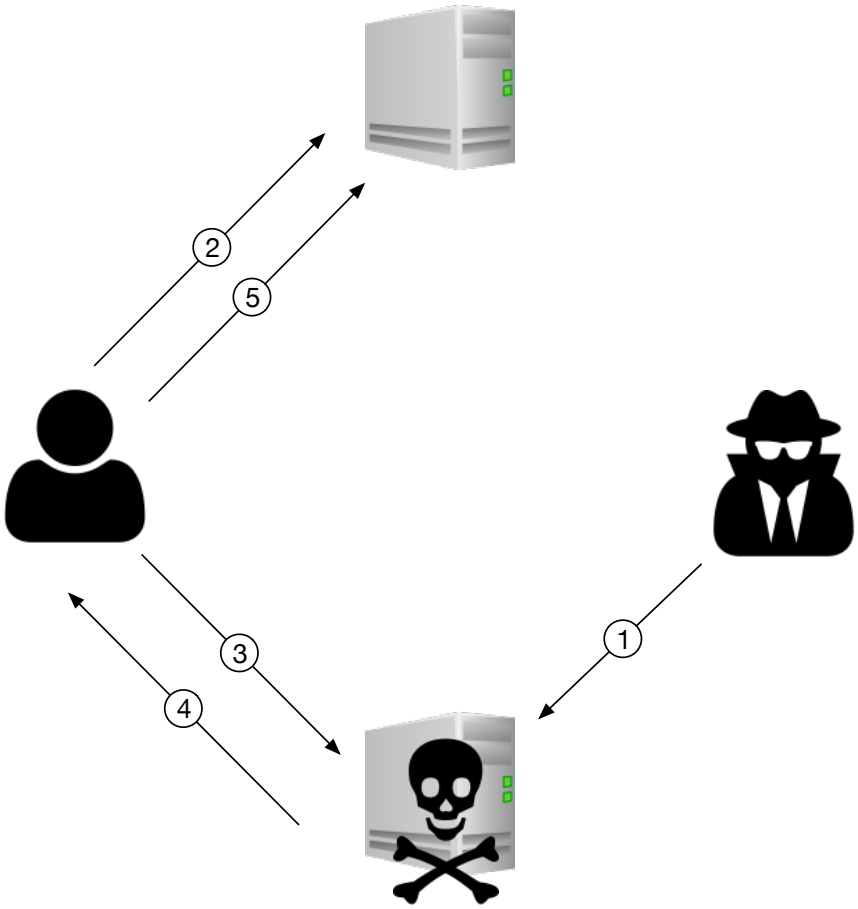


Fig. 2.5: Anatomy of a Cross-Site Request Forgery attack

Cookie, verifies that the correct token was sent within the request. Assuming

Table 2.1: Main problematic characters that could be used to perform XSS attacks and their corresponding HTML-encoding

Character	**HTML-encoding**
"	&quot;
'	&apos;
&	&amp;
<	&lt;
>	&gt;

the attacker can't determine the value of the token received by the user, he also can't forge the correct request that is accepted by the web application.

2.4 Conclusions

In this chapter, I have described the most dangerous vulnerabilities afflicting the security of web applications. All the vulnerabilities described in this section can be used to violate two security properties of interest: *authentication bypass* and *confidentiality breach*. For each vulnerability, I have described the exploitation techniques and the prevention techniques.

3

Software analysis

It is highly difficult to design and write software that does not contain bugs that might affect the intended execution of the software itself. Bugs in software may result in unexpected behaviors that can open the way to vulnerabilities that, as already described in Chapter 2, ultimately lead to compromise a system. Software analysis has thus become a crucial step in software development process to analyze the presence of known vulnerabilities. Software analysis can be divided in two main categories: *static analysis* and *dynamic analysis*.

Static analysis refers to the process of collecting information about a software without actually executing it. The information that are collected during the analysis depend on the requirements of the analysis itself and range from the identification of code that is never executed (also referred to as dead blocks) to the identification of tainted inputs that will inevitably result in injections vulnerabilities. Since the software is never really executed, the analysis does not depend on a specific input and thus the extracted properties are true for every execution of the software. However, a static analysis might not always terminate, or might require an a-priori unknown long time to terminate, due to the complexity of the analysis itself. Examples of static analysis techniques are:

- Abstract interpretation
- Model checking
- Symbolic execution
- Taint analysis

Opposed to static analysis, *dynamic analysis* is a type of software analysis where a software is executed on specific inputs and properties of the software are extracted by analyzing its output. In contrast with static analysis, the properties derived from a dynamic analysis are true only for the particular execution being observed. Dynamic analysis requires a large set of test cases to use as input for the software in order to cover all (or at least the majority) of the implemented behaviors. Examples of dynamic analysis techniques are:

- Unit testing
- Penetration testing
- Model-based testing

In the contest of this thesis, I will describe static and dynamic analysis techniques that are closely related to my research and that inspired the analysis I propose. More specifically, I will describe model checking as a static analysis technique and penetration testing and model-based testing as dynamic analysis techniques.

3.1 Static analysis

In this section I describe Model checking for statically analysing software and I describe the AVANTSSAR platform [6] which I use as back end of the analysis I propose in this thesis.

3.1.1 Model Checking

Model checking is a formal verification technique for assessing whether a property holds on system. A model checking analysis requires a formal specification of the system (called system model) along with a property of interest that should be analyzed on the system. The system model describes how the system behaves while the property of interest describes what the system model should do or what it should not do. The system model can be automatically generated from the source codeof a software written in certain programming languages like C or Java or can be generated by hands when automatic generation cannot be applied. Properties to analyze depend on the requirements on the system itself, for example: can Alice authenticate Bob and vice versa, agreeing on two secret nonces? The system model and the property of interest are then given to a model checker, which explores all the possible states of the system in a brute-force manner. The aim of such systematic exploration is to find a state that violates the property defined on the model. If, during the exploration, the model checker encounters a state that violates the given property, it provides a counterexample that indicates how the model can reach the undesired state. The counterexample describes an execution path that starts in the initial state and leads to a final state where the property is violated. Model checking is a general approach and can be applied in many different areas ranging from hardware verification to network security protocols verification.

In the context of this thesis, I am interested in model checkers for network security protocols verification formalizingthe Dolev-Yao (DY) [27] intruder model. A model checker implementing the DY intruder model is a specific type of model checker that implements a network and allows the definition of entities that can communicate with each other by sending messages over

communication channels. Furthermore, a model checker implementing the DY intruder, assumes that the network is under the control of an active intruder (the so called DY intruder) who has control of the entire network and can perform some actions on the messages that transit on it. In particular, the DY intruder can read and destroy any message, modify and create new messages as long as he does not break cryptography (following the perfect cryptography assumption).

3.1.2 The AVANTSSAR platform

In this section I briefly describe the AVANTSSAR platform [6] and the ASLan++[85] formal language used to model network protocols.

The AVANTSSAR platform (Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures) provides a comprehensive set of tools and formal languages for the specification and automated validation of trust and security of Service-Oriented Architectures (SOAs) and, in general, applications in the Internet of Services. The platform takes as input a formal specification of a Service-Oriented architecture and, using model checking techniques, generates an Abstract Attack Trace (AAT) which represents an high-level sequence of actions that violates a security property defined on the specification. The AVANTSSAR platform supports different specification languages such as BPMN, ASLan [10] and ASLan++ [85]. The core of the AVANTSSAR platform comprises three back-ends which, by operating on the same input, analyse the specification. Specifically, the AVANTSSAR platform comprise the three back-ends CL-AtSe [80], OFMC [14] and SATMC [8].

Figure3.1 shows the architecture of the AVANTSSR platform as described in [6] which is divided in two main layers: the connector layers and the validation layer. As already stated, the input of the AVANTSSAR platform is a specification of the services to analyse written in one of the supported formal languages. The connector layer provides a set of softwares that translate the input to the AVANTSSAR platform into the low-level language ASLan which is the main language supported by the validation layer. The validation layer provides a set of softwares for the formal analysis of the ASLan specification coming form the connector layer. More specifically, the input of the validation layer is a specification of the services to analyse along with a policy stating the security requirements of the input services. The orchestration phase of the validation layer analyses the ASLan specification looking for a composition of the services that is expected but not yet guaranteed to satisfy the security requirements of the input policy. The orchestrator generates a specification of the services that is guaranteed to satisfy the security requirements. The validator then takes the output of the orchestator and checks whether it satisfies the security requirements. If that is the case, the orchestrated service is returned as output, otherwise, a counterexample is generated and given to the orchestrator phase to provide a different orchestration. This process is

repeated until a valid orchestration is generated, or no suitable orchestration can be found.

The AVANTSSAR platform can also be used manually where, the orchestration phase is not used and the user manually generates the target service and security requirements to input to the validator. The validator analyzes the input and generates either a success (i.e, the input satisfies the security requirements) or a counterexample in for of an AAT.

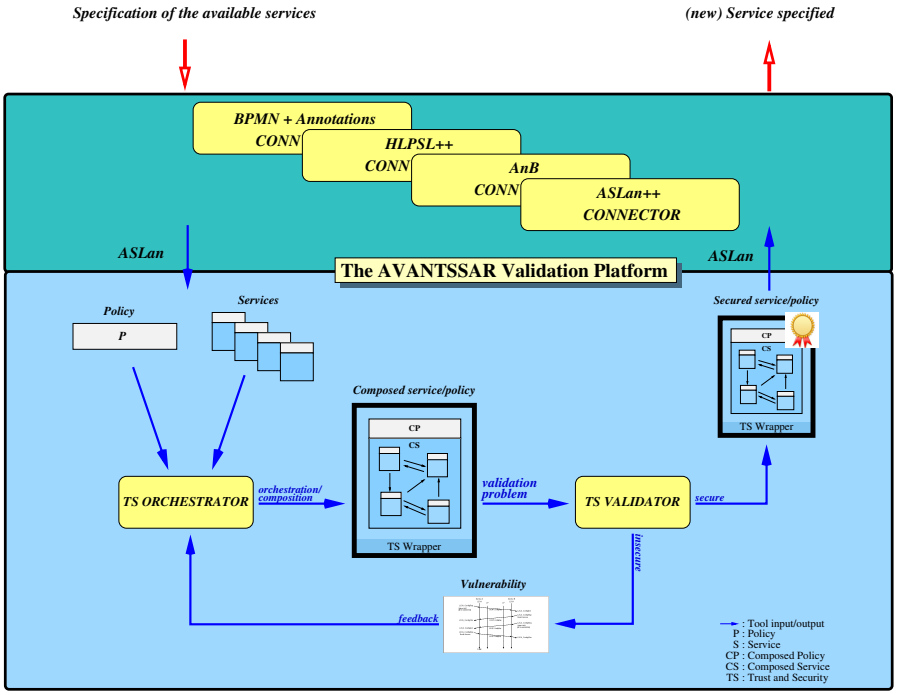


Fig. 3.1: The AVANTSSAR validation platform

3.1.2.1 ASLan connector

The ASLan connector is a module written in Java that translates high level input languages such as BPMN and ASLan++ into the low-level language ASLan. The purpose of this module is to allow security analyst to write specifications in a more easy to use language since the definition of complex services at the ASLan level is not practically feasible and can become quite cumbersome. It is important to underline that only syntactic translations is performed by the connector preserving the semantics of the model. The ASLan connector is also used to translate from ASLan to Message Sequence Chart (MSC)

in Alice-and-Bob notation. This is particularly helpful in the case the validator generates a counterexample so it can be easily reviewed by the security analyst.

3.1.2.2 The formal language ASLan++

ASLan++ [12] is a formal and typed security protocol specification language whose semantics is defined in terms of the ASLan language [11]. ASLan++ and ASLan are used by the AVANTSSAR [6] platform and the SPaCIoS Tool [84]. I won't go into the full details of the ASLan++ language, but in the following I discuss the main aspects of the ASLan++ language needed to understand my formalization.

Specification

The ASLan++ language resemble an Object-Oriented programming language in the way it defines its main structure. An ASLan++ specification consists in a hierarchy of *entity declarations*, which are similar to Java classes. The top-level entity is usually called *Environment* and it typically contains the definition of a *Session* entity, which in turn contains a number of sub-entities that define the main principals involved in the communication (e.g., web application, database). Each sub-entity defines the internal behavior of the component it models and the interaction with other entities. The definition of an entity starts with the keyword `entity` followed by the name of the entity (beginning with a capital letter) and a list of parameters. Consider the example in Listing 3.1 where three entities are defined. The outer entity is called `Environment` (line 1) which contains a sub-entity `Session` (line 2), which in turn contains two more sub-entities `Webapplication` (line 3) and `Database` (line 4). Entities may also have parameters which are defined in brackets. In the case of the entity `Webapplication` there are two variables, `Actor` and `Database` of type `agent`.[1] In this case, `Database` is a normal variable while `Actor` is actually a keyword used to represent the current entity. The keyword `Actor` can be compared to the keywords **this** in Java or self in Python.

Listing 3.1: Example of ASLan++ entities

```
1 entity Environment{
2  entity Session{
3    entity Webapplication(Actor, Database: agent):{}
4    entity Database(Actor, Webapplication: agent):{}
5  }
6 }
```

The content of an entity is composed by five main sections:

- `type`, in which custom data types can be defined,

[1] See § 3.1.2.2 for details on ASLan++ data types.

- `symbols`, in which variables, constants, functions and predicates are declared,
- `clauses`, where Horn clauses for that entity can be defined,
- `body`, where the behavior of the entity is defined. The instantiation of an entity must be done in the body of the parent entity using the `new` keyword as follows: `new subentity(<params>)`,
- `goals`, where security properties can be defined for the entity.

In ASLan++ comments start with the percentage character `%` which is used to comment a single entire line.

Data Types

In an ASLan++ entity, a section called `type` is reserved to the definition of custom data types which actually are sub-types of basic data types. Before detailing sub-types, I describe basic data types. ASLan++ supports the following basic data types:[2]

- `agent`: a communicating entity;
- `text`: an atomic message and is used for all those values that may be sent over the network
- `message`: a non atomic message;
- `fact`: a boolean value.

A sub-type defines a data type can be used when another data type is expected. For example Listing 3.2 defines a custom type `mytype` which is subtype of type `message` meaning that `mytype` can be used when a type `message` is required, but a type `message` cannot be used when a type `mytype` is required.

Listing 3.2: ASLan++ example of a custom sub-type

```
1  % Declaring sub-type
2  mytype < message;
```

Horn clauses

In an ASLan++ entity, a section called `clauses` is reserved to the definition of Horn clauses. A Horn clause is of the form `HCname: head :- body;`.

Symbols

In an ASLan++ entity, a section called `symbols` is reserved to the declaration of *variables*, *constants*, *sets*, *functions* and *predicates*.

[2] See [12] for the full list of supported data types.

Variables and constants

Variables in ASLan++ start with an upper case character (line 2 in Listing 3.3) while constants start with a lower case character (line 4 in Listing 3.3). Variables assignments are of the form `Var:=m` where `Var` is a variable and `m` is a constant. It is also possible to assign a random fresh value (i.e., a random value that has never been used before) by using the function `fresh()` (e.g. `Var := fresh()`). Finally, it is sometime useful to allow two entities to have access to the same data, meaning that they share some data. This can be achieved by defining *global variables*. A global variable is declared in the outermost entity so that it can be used in any inner entity.

Sets

ASLan++ supports sets of elements of a certain type with the keyword `set` (e.g. `myset : message set` defines a set of type `message`). Three basic operations are defined on sets that can be used to query and manipulate sets:

- **Contains:** `MySet->contains(E)` is used to query `MySet` on whether the element `E` belongs to `MySet`. The character `?` can be used to return an element that belongs to the set. For example `MySet->contains(?E)` stores in the variable `E` a random element contained in `MySet`, if there is at least one element in the set.
- **Add:** `MySet->add(E)` is used to add element `E` to `MySet`.
- **Remove:** `MySet->remove(E)` is used to remove element `E` from `MySet`.

Functions

In ASLan++ there is no functions call and functions do not have a body but are represented as uninterpreted functions. A function starts with a lower case character, defines a number of parameters and a return type (line 6 in Listing 3.3).

Predicates

Predicates are defined in the same way functions are defined with the only exception that their return type is `fact` (line 8 in Listing 3.3). It is also possible to retract facts using the keyword `retract` (line 10 in Listing 3.3).

Listing 3.3: ASLan++ example representing variable, constant, function, fact and fact retraction

```
1  % Variable definition
2  Variable : message;
3  % Constant definition
4  constant : text;
5  % Function definition
6  fn (message) : text;
```

```
7  % Predicate definition
8  pred(message) : fact;
9  % Fact retraction
10 retract factName();
```

Body

In an ASLan++ entity, a section called `body` is reserved to the definition of the behavior of that entity which can be defined with a number of statements described in the following paragraphs.

Sending and receiving messages

Sending and receiving of messages are expressed in Alice-and-Bob notation: `A -> B: M;`, where `A` and `B` are entities and `M` a message. More formally, a *message send* statement, `Snd -> Rcv: M`, is composed by two variables `Snd` and `Rcv` representing sender and receiver, respectively, and a message `M` exchanged between the two parties. In *message receive*, in order to assign a value to the variable `M`, a `?` precedes the message `M`, i.e., `Snd -> Rcv: ?M`. The ASLan++ keyword `Actor` refers to the entity itself (similar to "this" or "self" in Object-Oriented languages) and thus we actually write the send and receive statements as `Actor -> Rcv: M` and `Snd -> Actor: ?M`, respectively. `?` acts as a wildcard if it is not followed by any variable (e.g., `Snd -> Actor: ?`) since no specific pattern of the receiving message is expected. When defining the communication between two entities, it is possible to define different kind of communication channels having different kind of properties. The channels types supported by ASLan++ are summarized in Table 3.1 along with the security properties they ensure.

Table 3.1: Channel types supported by ASLan++

ASLan++ symbol	**Explanation**
`->`	Insecure communication channel.
`*->`	Authentic communication channel.
`->*`	Confidential communication channel.
`*->*`	Authentic and confidential communication channel.

`if` statement

The `if` statement is a control flow statement that allows to specify two possible behavior based on a boolean guard. The syntax (Listing 3.4) and semantics of an `if` statement are quite straightforward and comparable to common programming languages. The guard of the `if` can be any expression of type `fact`. In the case the guard is true, the true branch is evaluated, otherwise the else branch is evaluated.

Listing 3.4: ASLan++ example of an `if` statement

```
1 if(<guard>){
2   % true branch
3 else{
4   % false branch
5 }
```

`while` statement

The `while` statement is a control flow statement that allows to specify loops. The syntax (Listing 3.5) and semantics of the `while` statement are quite straightforward and comparable to other programming languages. The guard of the `while` can be any expression of type `fact`. If the guard evaluates to true, the body of the while is executed, otherwise the next statement after the while is executed. The `while` statement can be used in ASLan++ to simulate an entity that is actively listening for incoming connections. This is done by specifying the guard of the while loop to the constant `true` which always evaluates to the logical value *true* thus defining an endless loop.

Listing 3.5: ASLan++ example of an `while` statement

```
1 while(<guard>){
2   % while loop
3 }
```

`select-on` statement

ASLan++ includes a statement called `select-on` that can be compared to the `if` statement in the sense that it allows to define a control flow based on the value of some guards. There are, however, two important differences: (1) more than one guard can be defined (2) when evaluating the guards of a `select-on` statement, if none is valid, the execution is blocked until one becomes valid, meaning also that there is no fail branch. The syntax of the `select-on` statement (Listing 3.6) starts with the keyword `select` which encloses one or more `on(<guard>)` where guard can be any expression of type `fact`. The body of a `select-on` branch is enclosed in curly brackets. Since the evaluation of a `select-on` might block the execution of the model in the case no guard evaluates to true, one might ask what happen in the case that more then one guard evaluates to true. In this case, the model checker will non-deterministically execute one of the branch for which the guard is true. The `select-on` is thus a statement that allows to introduce non-deterministic behavior in an ASLan++ specification.

Listing 3.6: ASLan++ example of a `select-on` statement

```
1 select{
2   on(<guard1>):{
```

```
    % body in case guard1 is true
    }
    on(<guard2>):{
    % body in case guard2 is true
    }
    ...
}}
```

Goals

In an ASLan++ entity, a section called `goals` is reserved to specify security properties that have to be checked on the specification. In general, the goals section is only defined in the `Session` entity defining security properties for its sub-entities. In the `goals` section one can specify LTL formulas to use as goals such as `[]pred(Var)` stating that the predicate `pred` must always hold over a variable `Var`. The LTL operators needed to understand the case studies are shown in Table 3.2.[3]

Table 3.2: Subset of the LTL operators supported by ASLan++

Operator	ASLan++ syntax	Explanation
$\neg$	`!f`	negation
$=$	f1 = f2	equality
$\neq$	`f1 != f2`	inequality
$\wedge$	`f1 \& f2`	conjunction
$\vee$	`f1 | f2`	disjunction
Globally	`[](f)`	always

3.1.2.3 From ASLan++ to ASLan

An ASLan++ specification can be automatically translated (see [6]) into a more low-level ASLan specification, which ultimately defines a transition system $M = \langle \mathbf{S}, \mathbf{I}, \rightarrow \rangle$, where $\mathbf{S}$ is the set of states, $\mathbf{I} \subseteq \mathbf{S}$ is the set of initial states, and $\rightarrow \subseteq \mathbf{S} \times \mathbf{S}$ is the (reflexive) transition relation. The structure of an ASLan specification is composed by six different sections: signature of the predicates, types of variables and constants, initial state, Horn clauses, transition rules of $\rightarrow$ and protocol goals. The content of the sections is intuitively described by their names. In particular, an initial state $I \in \mathbf{I}$ is composed by the concatenation of all the predicates that hold before running any rule (e.g., the agent names and the attacker's own keys). The transition relation $\rightarrow$ is defined as follows. For all $S \in \mathbf{S}$, $S \rightarrow S'$ iff there exist a rule such that

[3] See [12] for the full list of supported LTL operators.

$$PP.NP\&PC\&NC \Rightarrow\!\!\lceil V \rceil\!\!\Rightarrow R$$

(where PP and NP are sets of positive and negative predicates, PC and NC conjunctions of positive and negative atomic conditions) and a substitution $\gamma : \{v_1, \ldots, v_n\} \to T_\Sigma$ where $v_1, \ldots, v_n$ are the variables that occur in PP and PC such that: (1) $PP\gamma \subseteq \lceil S \rceil^H$, where $\lceil S \rceil^H$ is the closure of S with respect to the set of clauses H, (2) $PC\gamma$ holds, (3) $NP\gamma\gamma' \cap \lceil S \rceil^H = \emptyset$ for all substitutions γ' such that $NP\gamma\gamma'$ is ground, (4) $NC\gamma\gamma'$ holds for all substitutions γ' such that $NC\gamma\gamma'$ is ground and (5) $S' = (S \setminus PP\gamma) \cup R\gamma\gamma''$, where γ'' is any substitution such that for all $v \in V$, $v\gamma''$ does not occur in S.

I now define the translation of the ASLan++ constructs I have considered here. Every ASLan++ entity is translated into a new *state predicate* and added to the section signature. This predicate is parametrized with respect to a *step label* (that uniquely identifies every instance) and it mainly keeps track of the local state of an instance (current values of whose variables) and expresses the control flow of the entity by means of step labels. As an example, in the case of the ASLan++ entity

```
entity Snd(Actor, Rcv: agent){
 symbols
    Var: message;
}
```

then the predicate `stateSnd` is added to the section signature and, supposing an instantiation of the entity `new Snd(snd, rcv)`, the new predicate `state_Snd(snd, iid, sl_0, rcv, dummy_message)` is used in transition rules to store all the informations of an entity, where the ID `iid` identifies a particular instance, `sl_0` is the step label, the parameters `Actor`, `Rcv` are replaced with constants `snd` and `rcv`, respectively, and the message variable `Var` is initially instantiated with `dummy_message`.

Given that an ASLan++ is a hierarchy of entities, when an entity is translated into ASLan, this hierarchy is preserved by a `child(id_1, id_0)` predicate that states `id_0` is the parent entity of `id_1` and both `id_0` and `id_1` are entity IDs.

A variable assignment statement is translated into a transition rule inside the rules section. As an example, if in the body of the entity `Snd` defined above there is an assignment `Var := constant;` where `constant` is of the same type of `Var`, then the following transition rule is obtained:

```
state_Snd(Actor,IID,sl,Rcv,Var)
=>
state_Snd(Actor,IID,succ(sl),Rcv,constant)
```

In the case of assignments to `fresh()`, the variable `Var` is assigned to a new variable.

In the case of a message exchange (sending or receiving statements), the `iknows(message)` predicate is added to the right-hand side of the corresponding ASLan rule. This states that the message `message` has been sent over the

network and `iknows` is used because, as is usual, the Dolev-Yao attacker is identified with the network itself.

The last point is the translation of goals focusing only on the LTL goal used in the case studies. Goals are translated into attack states containing the negation of the argument of the LTL operator:

```
attack_state authorization :=
 iknows(M)
```

More information on ASLan, ASLan++ and the AVANTSSAR Platform can be found in [12, 6].

3.1.2.4 Validators

The validators take as input an ASLan specification (which can be an orchestration or a manual specification) and check whether the ASLan specification meets the security requirements under the assumption that the network is controlled by a DY intruder. The three back-ends included in the AVANTSSAR platform, CL-Atse, OFMC and SATMC, provide the model-checking functionality for analyzing the ASLan specification. By default, all three back-ends run on the same specification so that the security analyst can compare results and performances of the three tools, however, the security analyst can also manually select one specific back-end to use.I now briefly describe the three back-end tools.

CL-Atse

The Constraint-Logic-based Attack Searcher (CL-AtSe) [80] is developed and maintained by the Institut National de Recherche en Informatique et Automatique. CL-Atse uses rewriting and constraint solving techniques to model all states that are reachable by the entities modeled in the specification and decides if the security requirements are violated with respect to the DY intruder. Moreover, it applies constraint solving with simplification heuristics and redundancy elimination techniques. CL-AtSe performs a bound analysis on the number of calls in case the specification allows for loops in execution and implements several preprocessing modules to simplify and optimize input specifications before starting a verification. In particular, it includes optimisations for step interleaving, either by preprocessing or by optimised data structures and deduction rules.

OFMC

The Open-source Fixedpoint Model Checker (OFMC) [14] extends the On-the-fly Model Checker (the previous OFMC). The two main techniques implemented in OFMC are the lazy intruder and constraint differentiation.

The lazy intruder is a symbolic representation of the intruder which aims at avoiding the naive search of the possible infinite search space generated

by a specification including a DY intruder. By using a symbolic, constraint-based approach, OFMC reduces the search space without excluding attacks and without introducing new ones.

Constraint differentiation integrates the lazy intruder and ideas from partial-order reduction technique to solve the problem resulting from the large number of possible interleaving due to parallel protocol executions. Constraint differentiation uses independence information from the symbolic transition system when reducing constraint and works by introducing a new kind of constraint.

SATMC

The SAT-based Model Checker (SATMC) [8] is an open, flexible platform for SAT-based bounded model checking of security services developed and maintained by University of Genova. SATMC reduces the problem of determining whether the system violates a security goal in $k > 0$ steps to the problem of checking the satisfiability of a propositional formula (the SAT problem). SATMC generates a formula that represents all the possible evolution of the transition system (up to k depth) described by the specification. Thus, finding an attack of length k is reduced to solving a propositional satisfiability problem. This SAT solving task is performed by state-of-the-art SAT solvers, which can handle propositional satisfiability problems with hundreds of thousands of variables and clauses. When SATMC finds a satisfiable formula, the corresponding model is translated back into a partially ordered set of rules that, starting from the initial state, will lead the specification into a state that violates the security requirements defined on the specification.

3.2 Dynamic analysis

In this section I describe two dynamic analysis approaches that provide the basis of my research, namely: penetration testing (§ 3.2.1) and model-based testing (§ 3.2.2). In particular, for penetration testing I will describe how security analysts perform this kind of testing and the tool suite that istypically used for penetration testing of web applications. For model-based testing, after giving a high level description of the approach, I will describe the SPaCIoS tool from which I started to develop the research proposed in this thesis.

3.2.1 Penetration Testing

The most used methodology to assess the security of web applications (or software in general) is *penetration testing*. Penetration testing (or informally *pentesting* is used to describe an activity in which an authorized security analyst simulates attacks on a system with the intent of finding weaknesses that may cause harm to the system itself. Penetration testing issometimes,

confused with, or used as synonymous of, *vulnerability assessment*. However, there is a fundamental difference between a vulnerability assessment and a penetration test: a vulnerability assessment is meant to identify vulnerabilities while a penetration test is meant to exploit vulnerabilities to actually harm the system. The goal of a penetration test is to actually identify what sort of damages a real attacker might cause to a system with respect to a security property. The security property depends on the system itself, an example could be: is it possible, for unauthorized users, to have access to resource X? It might also be used to assess the ability of the environment surrounding the system to successfully detect and respond to attacks. and a specific security property to assess. An important aspect of penetration testing is to conduct the tests in a response way. The tests need to be scheduled and planned in such a way that they do not cause any denial of service or real harm to the system. Finally, once the tests are completed, the security analyst writes a summary report to the owner of the system describing his findings so that appropriate countermeasures can be taken.

To perform a penetration test, security analysts make use of a number of automatic tools. However, automatic tools are not yet capable of entirely replacing the human factor [29] meaning that the experience of a penetration tester plays a central role

3.2.1.1 Penetration testing methodologies

There are two main approaches used to conduct a penetration test: *black-box* and *white-box* testing. The main difference between black-box and white-box is the amount of knowledge of a system under analysis System Under Test (SUT) that is available to the security analyst prior to the beginning of the penetration test. The two approaches can be used to analyze a system from two different points of view.

Black-box testing assumes that very little knowledge of a system is available before starting the test. The security analyst starts with a very small set of information such as an URL or an IP address and from that he has to extend his knowledge and ultimately compromise the target system. Black-box testing simulates an attack coming from an external attacker and thus unfamiliar with the environment surrounding the SUT. Black-box analysis can be used to understand what sort of information an attacker might be able to collected and use to harm the SUT.

Opposite to black-box, *white-box* testing provides the security analyst a complete knowledge of the SUT. The initial knowledge might include anything form network diagrams, source code and IP addresses information. White-box testing simulates what might happen during an attack performed from insiders or as a result of the leak of sensitive information, where the attacker gets access to source code, network layouts, and possibly even some passwords.

In most situations, however, black-box and white-box can complement and enhance each other. For example, having identified some anomalous behavior

during a black-box analysis, the easiest way to investigate its root cause is to review the source code of the system behaving anomalously.

Finally, even tough black-box and white-box penetration testing are generally very effective in finding threats to a system, they lack any rigorous or formal description. There has been different attempts in identifying a precise methodology to perform a penetration test such as collections of vulnerabilities, tools and check lists have been proposed to be used as guidelines for performing penetration testing. In particular, it is relevant to name a few: OWASP [60], OSSTMM [40] and CAPEC [51].

OWASP

The Open Web Application Security Project has developed a widely used set of standards, resources, training material, and the famous OWASP Top Ten list mentioned in Chapter 2 which is mainly focused on the security of web applications.

OSSTMM

The Open Source Security Testing Methodology Manual is a widely used methodology that covers all aspects of performing a penetration test. The purpose of the OSSTMM is to develop a standard that, if followed, will ensure a baseline of tests to perform, regardless of customer environment or test provider. This standard is open and free to the public, as the name implies, but the latest version requires a fee for download.

CAPEC

The Common Attack Pattern Enumeration and Classification (CAPEC) aims at providing a catalog of common attack patterns to wide range of systems along with a comprehensive description of requirements, limitations and related attacks.

3.2.1.2 Anatomy of a penetration test

This section gives an overview of the main steps followed by a security analyst when performing a penetration test.

Information gathering

During this step, the security analyst tries to get as many information as possible of it's target. In order to do so, the security analysts takes advantage of freely available resources (search engine, social networks etc.) where he might find public useful information. The activity of online searching and collection of information is usually known as Gathering Open Source Intelligence (OSINT). Additionally, information can be collected by the target itself, to

do so the security analyst employees tools such as port scanners[4] to collect information on the target system such as as what software is running. This process may not be necessary when performing white-box testing but it surely is necessary when performing black-box testing.

Vulnerability assessment

Next, the security analyst begins the discovery of vulnerabilities. Vulnerabilities are the means attackers use to successfully compromise a system. In this phase a security analyst makes extensive use of vulnerability scanners. A vulnerability scanner is a software which purpose is to identify a set of known vulnerabilities on a system. A vulnerability scanner employees vulnerabilities databases and active checks to best identify which vulnerabilities are present on a system. Modern vulnerability scanners are very powerful tools capable of identifying many vulnerabilities, however, they can't fully replace the human factor [29], so it is also required to perform manual analysis and verify the results.

Exploitation

The previous two steps give an overview of the kind of information disclosed by the system and the vulnerabilities it is affected by. However, these information are not sufficient to truly understand the level of security of a system. This is why the exploitation phase is an important step in the penetration test. The security analyst has to use his creativity and experience to create exploits using the information and vulnerabilities found in the previous steps in an attempt to access the target system. Some vulnerabilities are remarkably easy to exploit, such as logging in with default passwords while others might require more sophisticated actions, such as writing a custom code to exploit a vulnerability in a specific programming language.

Post exploitation

The post exploitation step, as the name itself suggests, starts once an exploitation was successful and the security analyst has now accessed to the target system. In post exploitation, the security analyst gathers information about the attacked system and tries to further compromise other systems and collect more information. It might be possible, for example, to dump password hashes and attempt an off-line brute force attack and, if a password was reversed, it might be used to access other systems.

Report

Finally, a penetration testing is of no use if the results of the analysis is not reported to the owner of the attacked system and then fixed. The security

[4] See § 3.2.1.3 for more details on the tools used during a penetration test.

analyst should report the information related to his findings during the entire penetration testing. detailing the weaknesses he identified and how he successfully exploited such weaknesses to compromise the system. Writing a good report is as import, and difficult, as any of the previous steps might be.

3.2.1.3 Toolkit

In this section I list and describe the main tools used in a penetration testing session. This classification is not exclusive but it covers the vast majority of the security tools nowadays available.

Web Applications Proxies

The best way to understand the communication "under the hood" with a web application is to use an HTTP proxy. An HTTP proxy is a tool capable of intercepting an HTTP request (and its response) and gives the possibility of showing or editing the request (or the response) before it gets sent (or received). An HTTP proxy is the most useful tool when it comes to analyzing web applications. Their flexibility allows the security analyst to easily perform basic malicious injection test and analyze the behavior of a web application to such malicious input. Examples of well known and still active HTTP proxies are:

- Burp Proxy [68]
- Fiddler [78]
- OWASP Zed Attack Proxy [62]

Vulnerability scanners

A vulnerability scanner is a software whose purpose is to identify a set of known vulnerabilities on a system. A vulnerability scanner employees vulnerabilities databases and active checks to best identify which vulnerabilities are present on a system. Vulnerability scanners are generally able to identify the target operating systems to decide which subset of the known vulnerabilities should be tested. They are usually not able to probe general purpose applications, and are not capable of identifying unknown vulnerabilities. Some vulnerability scanners are able to exploit network links with other sub-systems by recursively scanning the hosts on the targeted network. Vulnerability scanners also differentiate on the target they analyze, being it a stand alone application, an operating system, a web application or whether they search for a specific vulnerability such as SQL Injection, Cross-Site Scripting or buffer overflows. In my thesis I am mainly interested in the security of web applications and thus I report only examples of well known vulnerability scanners used to analyze web applications:

- Nikto [77]

- Arachni [5]
- W3af [69]
- Metasploit [46]
- Acunetix vulnerability scanner [1]
- OpenVAS [36]
- Netsparker [54]
- Sqlmap [74]
- Wfuzz [88]
- Dotdotpwn [28]
- DIRBuster [58]

3.2.2 Model-Based Testing

As already mentioned, software testing aims at finding behaviors that differ from the one intended by the software. The main purpose is to find a failing path that leads the system to a state that was not expressed by its requirements and in security testing this translates in finding vulnerabilities. One issue of a totally manual testing approach, is the identification of meaningful tests to perform on the SUT. Model-based testing (MBT) [33] is a software testing technique which aims at providing a solution to the generation of meaningful test cases. MBT employees the use of behavioral models representing the intended behavior of the SUT. Tests, namely pairs of inputs and expected outputs, are generated from the behavioral models and are then performed on the real SUT. A model describing the SUT is, generally, an abstract representation of the real SUT where some details are not represented. As a result, the tests generated are at the same abstraction level as the model itself and are called Abstract Attack Traces (AATs). An AAT cannot be performed on the real SUT at its level of abstraction but it requires an additional step of concretization where the AAT is mapped to concrete actions (such as the execution of an HTTP request) that can be performed on the SUT.

MBT tries to bring together the powerfulness and reliability of formal methods and the effectiveness of testing. Test cases with a specific purpose can be automatically generated thanks to the capability of back-end formal methods such as model-checking. In fact, it is possible to formalize a test case in terms of a specification goal that is then used with the model of a SUT in order to treat test case generation as a model checking problem.

A general representation of MBT approach is shown in Figure 3.2 where they main steps for the generation and execution of test cases are:

1. A behavioral model is generated from the SUT. The generation of such a model can be performed manually or automatic tools could be used when available;
2. The behavioral model is then given in input to the Tests Generation Engine (Test Generation Engine (TGE)) which analyzes and generates tests. There are a variety of ways in which tests can be generated depending on the model itself (e.g.: model-checkers, SAT solvers, etc.);

3. AATs are generated from the TGE and are fed to the Tests Execution Engine (TEE);
4. The TEE is in charge of concretizing the AATs and to test them on the SUT. When a test if generated, the TEE decides either automatically or by manual intervention whether the test was successful or not;
5. The result of a tests can be used to further refine the behavioral model.

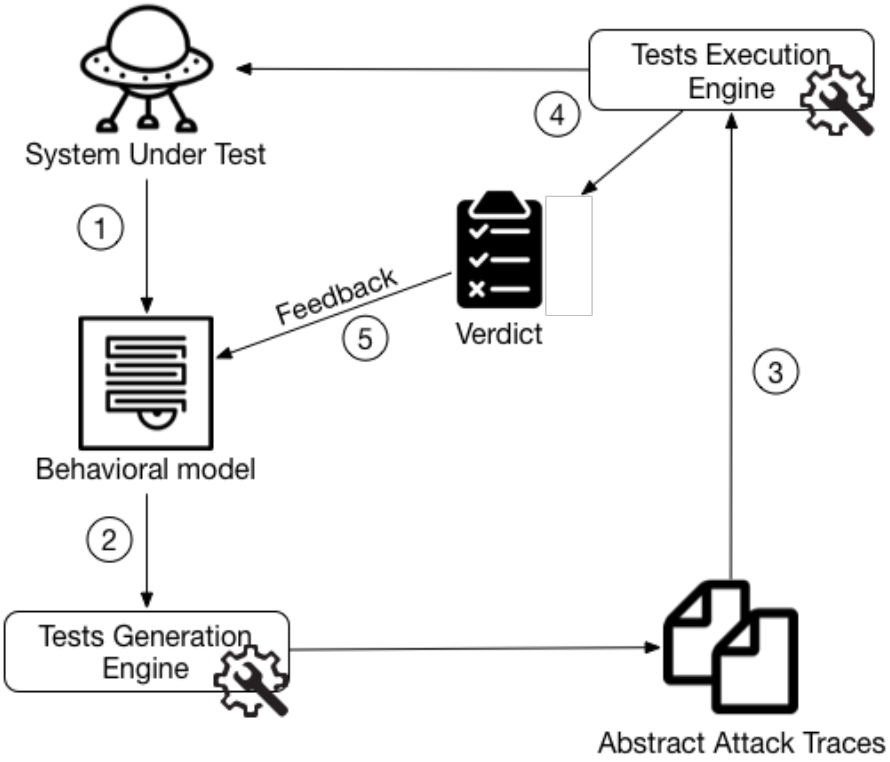


Fig. 3.2: General representation of a Model-Based Testing approach

3.2.3 The SPaCIoS tool

Secure Provision and Consumption in the Internet of Services (SPaCIoS) is a collection of tools related to modeling, verification and testing of internet services (which includes, but not limited to, web applications and network protocols). The SPaCIoS tool shows the effectiveness of applying formal methodologies such as model checking to the runtime analysis of the security of internet services.

The implementation of the SPaCIoS tools consists of a collection of different plug-ins for the Eclipse IDE providing a MBT solution for testing internet services within the same environment used to develop such services. Figure3.3 shows the main components of the SPaCIoS tool which resembles the general approach of MBT shown in Figure3.2. The security analyst interacts with the user interface of the SPaCIoS tool which is provided by the Eclipse workbench and workspace. The workbench contains the main graphical components of the SPaCIoS tool while the workspace contains resources and data needed by the SPaCIoS tool. The user interface of SPaCIoS is integrated and extends the Eclipse workbench with new commands used to control all the tools implemented by SPaCIoS and the editors used to provide new input to the SPaCIoS tool. The creation of the formal model is performed within Eclipse itself thanks to the editors provided by the SPaCIoS tool. Once the formal model of a SUT and a security property have been created, the security analyst can invoke the *Property-driven and vulnerability-driven test generation*. This component generates the AATs using the model of the SUT and the security property of interest. The test case generation component may also take advantage of a *trace-driven fault localization* which is based on the source code of the SUT, when available. The test cases generated represents an execution trace of the SUT leading it to a state where the security property is violated. The *Libraries* component of SPaCIoS comprises four different sets: vulnerabilities, attack patterns, security goals and attacker models. These sets are used as input for the *property-driven and vulnerability-driven test case generation* component as well. Attack patterns are also used to guide the security analyst in the iterative testing phase carried out by the *Test Execution Engine* (TEE), which is in charge of executing the test cases by handling the communication with the SUT. However, a test might fail revealing a discrepancy between the model of the SUT and the real SUT which means that some adjustments should be performed on the model.

The main modeling language employed in the SPaCIoS tools is ASLan++. As I already stated in § 3.1.2.2, ASLan++ is suitable for the definition of distributed systems that exchange messages over a different range of secure channels and provides the flexibility to define a variety of security goals. ASLan++ facilitates the specification of internet services at a high level of abstraction in order to reduce model complexity as much as possible, and at the same time is close to procedural and object-oriented programming languages, so that it can be employed by security analysts who are not formal specification experts.

The SPaCIoS tools allows security analysts the flexibility to select which tool should be executed in isolation and the possibility to follow different MBT flows.

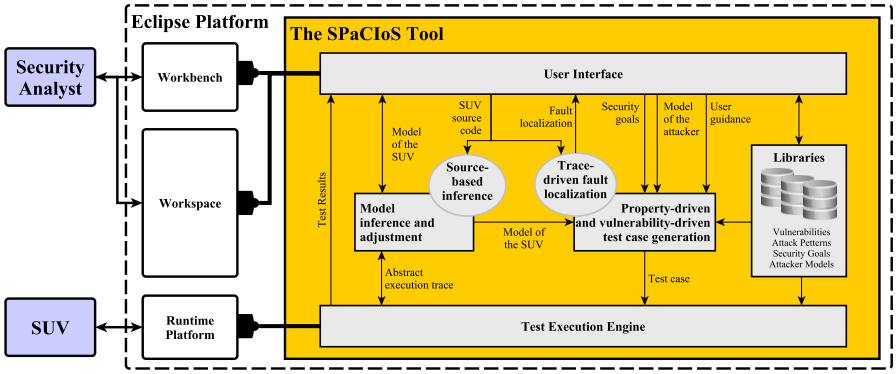


Fig. 3.3: The SPaCIoS tool and its main components

3.3 Conclusions

In this chapter, I have described the main software analysis techniques that are used nowadays to analyze the security of a system. I have described static and dynamic analysis techniques with emphasis on the secure analysis of web applications. More specifically, I have described model checking as a static analysis technique and the AVANTSSAR platform that uses model checking for analyzing the security of network protocols. Moreover, I have described penetration testing and model-based testing as dynamic analysis techniques, including in particular the SPaCIoS tool, which is used for analyzing the security of internet services.

Part II

Model-based Security Testing Framework (MobSTer)

4

MobSTer

In this part, I describe the *Model-based Security Testing Framework (MobSTer* that aims to support a security analyst in carrying out security testing of web applications. MobSTer was developed during the doctoral thesis of Dr. Michele Peroli in which I collaborated at the beginning of my Ph.D. During the development of MobSTer, I worked on the definition of actions, the basic building blocks of MobSTer that I describe in this chapter. Moreover, I performed comparative tests with state-of-the-art tools for the secure analysis of web applications in order to evaluate the effectiveness of MobSTer. The main idea underlying MobSTer is to combine model-checking techniques and penetration testing guidelines and checklists. The aim is to create a framework capable of searching for possible vulnerable "entry points" without missing important checks. More specifically, as shown in Figure 4.1, MobSTer follows the classical approach of Model-based testing. First, the security analyst creates a *model* of the web application by defining *actions*; an action, intuitively, is a part of the web application providing some particular functionality that can be accessed by users through a user interface or a web browser (e.g., authentication, user profile management or item purchase in an e-shopping application). The model is completed with a security property that should hold on the web application. The final model is then fed into a *Model Checker* that will return *Counterexamples (CE)* if any are found, which are execution traces that violate the security goal. However, both the actions themselves and the CEs are too abstract to be directly employed for testing the web application. MobSTer thus provides for a *Test Execution Engine (TEE)* that translates a CE into a sequence of *HTTP requests* that can be performed on the web application.

Actions are defined by the security analyst who decides the abstraction level and the granularity he wishes to consider. As a concrete example of the actions' characteristics (i.e., their use in the framework as well as the expertise used in their identification), consider the "login" functionality. Web applications usually employ one of the following types of authentication mechanism

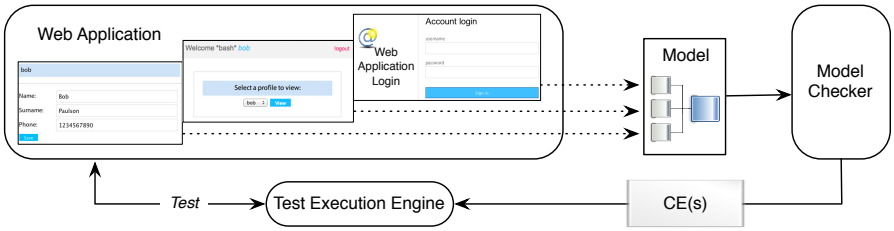


Fig. 4.1: A high-level view of the MobSTer framework

(it also includes the attacks that the security analyst can exploit in order to attack the action):

- *Basic Authentication*: the tester can try to perform a brute-force attack, and the password is sniffable if not passed via HTTPS.
- *HTTP Digest Authentication*: brute-force attacks and man-in-the-middle attacks are possible.
- *Form-based Authentication*: brute-force attacks are possible, injection vulnerabilities (e.g., SQL-injection [72], Cross-Site Scripting XSS [37], etc.) have to be tested, and the attacker should try to capture the authentication token (i.e., session tokens or cookies).

For each of these authentication types, a security analyst can create an associated action and also reuse it later to test different models.

MobSTer is implemented as a prototype[1] which was applied to test a number of case studies to assess its strength and concretely evaluate it with respect to four state-of-the-art tools that are normally used by penetration testers and that are close to what MobSTer aims to achieve: Burp Suite [68] (free version 1.7.23 and Pro version 1.7.23), OWASP Zed Attack Proxy [62] (ZAP, version 2.3.1) and Arachni [5] (framework version 1.5.1). Our evaluation shows that MobSTer has a better identification rate and a better vulnerability coverage than these four tools.

4.1 Modeling web applications for MobSTer

In MobSTer, the definition of models is inspired by the *HRU model*, a security model proposed by Harrison, Ruzzo and Ullman for the integrity of access rights in operating systems [39]. The HRU model defines (i) a finite set of generic rights R and (ii) a finite set C of commands containing conditions to be checked and primitive operations to be performed (i.e., create/destroy objects/subjects, give/delete rights on an object). A *configuration* of such a protection system is a triple (S, O, P), where S is the set of current subjects,

[1] The source code of the MobSTer tool is available at https://github.com/REGIS-lab/MobSTer.

O is the set of current objects, $S \subseteq O$, and P is an access matrix, with a row for every subject in S and a column for every object in O. $P[s, o]$ is a subset of R, the generic rights. $P[s, o]$ gives the rights to object o possessed by subject s. The "safety" problem for protection systems under this model is to determine in a given situation whether a subject can acquire a particular right to an object. Basically, safety means that an unreliable subject cannot pass a right to someone who did not already have it (i.e., the owner gives away certain rights to his objects).

To test web applications using MobSTer, an access matrix M like the matrix P of the HRU model is defined, where the commands are instantiated with the functionalities offered by the web application, the set of generic rights is redefined to express the information that permits one to relate web application functionalities to known vulnerabilities, and the set of primitive operations is changed according to the changes performed on the other concepts.

In MobSTer, the model of a web application for security testing is defined as a *transition system TS* that contains: (i) a data structure containing the data handled by the web application (and its users), (ii) the information about the storage and the management of this data by the web application, (iii) the functionalities that the web application provides, and (iv) how these functionalities can be accessed by the web application's users.

4.1.1 Users, data and knowledge

The set *UserName* is defined as the set of unique identifiers of users interacting with the web application, where a special label *Anon* is introduced for anonymous browsing. For the users in *UserName*, the security analyst has to establish which data is in the scope of the analysis.

Definition 4.1 (*UserData*). Let $MetaData$ be a set of abstract representations (defined by the security analyst) of the data implemented in web applications that a user can handle, $MetaData = n$ be the number of elements in $MetaData$, $BasicTypes = \{String, Int, Bool, \ldots\}$ be the set of concrete data types and $StrucTypes = \{Profile, Credential, \ldots\}$ the abstract types of the elements of $MetaData$. The record $UserData = (field_1, ..., field_n)$, where $field_i \in UserData$[2] is an instantiation (at some level of abstraction) of the i-th element of $MetaData$ and is of the form $field_i = [(subfield_{i.1}[, subfield_{i.2}], ...)]$, where

- $field_i$ has type in *StrucTypes* or *BasicTypes*, and
- $subfield_{i.j}$ are optional and have types in *BasicTypes*.

□

As an example, a typical data structure is

[2] With a slight abuse of notation, $\in$ is used also for records.

$$\begin{aligned} UserData = &(cred = (user, pwd),\\ &id,\\ &prof = (name, ...),\\ &messages = (m[, m]...[, m]),\\ &data = (d[, d]...[, d])). \end{aligned}$$

When a security analyst models a web application, some of the fields of *UserData* will remain unchanged, others will be modified (the choice will depend on the web application in some cases and on the modeling choices in other cases).

In a multi-user environment, every user has access to his (and other users') data through the interface of the web application. To model this aspect, the set *Data* contains the possible data that users can handle during their interaction with the web application.

Definition 4.2 (*Data*). The set *Data* is defined by instantiating every element in *UserData* with each user in *UserName*, i.e.,

$$Data = \{x.y \mid x \in UserName \text{ and } \ y \in UserData\}.$$

□

During the interaction with a web application, a user can access and use many types of knowledge. For instance, he can:

- use information he already knows, e.g., from the beginning of his execution (the origin of this initial knowledge is not discussed here and its definition is demanded to the security analyst),
- gain information from the interaction with the web application itself,
- or, in borderline cases, even guess some information.

These types of knowledge are described by means of labels contained in the set

$$K_{Source} = \{Initial, Gained, Guessed\},$$

where other sources of knowledge can of course be modeled by defining a different K_{Source}.

Definition 4.3 (Knowledge). The knowledge of the users is defined as a set of triplets

$$U_{Knows} = \{(x, d, k_{src}) \mid \ x \in UserName, d \in Data \text{ and } k_{src} \in K_{Source}\},$$

and $U_{Knows}^{s_i}$ denotes the content of the set U_{Knows} at state s_i.

□

4.1.2 The behavior of web applications

The behavior of a web application is modeled through events that a specific user triggers (i.e., causes to happen), through the use of the functionalities of the web application (i.e., actions), regarding some data, and with respect to a specific "location" on the server. In MobSTer, events describe what is happening to the web application's data. Events are related to actions in that when an action α is performed, some events take place (i.e., the user triggers some events through the use of the action).

Definition 4.4 (Events). Events describe how the data is managed by the information technology that the web application relies on (e.g., databases that manage data, file systems for files, sessions for access control and volatile data, operating systems that execute commands, and the web application's user interface that retrieves data from the users). The syntax for defining *events* is

$$x.event(parameters, location)\,,$$

where $x \in UserName$, *event* is the event's name, $parameters \subseteq Data$ and *location* is the technology used by the web application to handle the data (e.g., the file system).

□

For instance, if an action models a functionality that allows a user to write some data on the web application's database, then the event $x.write(targetData, database)$ is related to that action.

Actions (Definition 4.9) are used to move the transition system from a state (Definition 4.8) to another. In order to simplify the notation for the actions, events are specified in the target state of the action, i.e., for states s_i and s_{i+1} of the transition system,

$$s_i \xrightarrow{\alpha_i+\{events\}} s_{i+1} \quad \text{becomes} \quad s_i \xrightarrow{\alpha_{i+1}} s'_{i+1},$$

where in s'_{i+1} for each event in $\{events\}$, a label expressing the *event* and its *location* is specified in the event's *parameters*. In those cases in which it is not possible to determine the location where the data are managed, the security analyst can create multiple models with different guesses, and test each such model. The automatic testing process will take care of reducing the overhead of testing multiple traces.

Similar to the definition of the knowledge of the users, the set *Event* contains the possible labels for events. As a concrete example, a useful instantiation of the set *Event* (that could, of course, also be modified by the security analyst) is

$$Event = \{ShowDB, WriteDB, ShowFS, WriteFS, Exec, WriteSD, ShowSD\}\,,$$

where the intended meanings of the labels in *Event* are:

- *ShowDB*: the labeled data has been displayed as a result of a query that reads from a database (e.g., the messages in an online forum).
- *WriteDB*: the labeled data has been written in a database (e.g., if a user saves his profile on a database).
- *ShowFS*: the labeled data has been read from the file system and displayed (e.g., the web application has a photo album whose photos are read from files).
- *WriteFS*: the labeled data has been written on the file system of the server (e.g., photos, attached documents, etc.).
- *Exec*: the labeled data has been displayed and retrieved as part of the execution of a command (e.g., the open function in PERL or the *Runtime class* in Java).
- *ShowSD*: the labeled data has been retrieved and displayed from a local session of the browser (e.g., preferences or runtime state).
- *WriteSD*: the labeled data has been saved in the browser along with the other data pertaining to a certain session.

Definition 4.5 (WA_{Event}). The set of events that are related to the user and the data is defined as:

$$WA_{Event} = \{(x, d, e) \mid \ x \in UserName, d \in Data \text{ and } e \in Event\},$$

where a triplet in WA_{Event} states that the event e happens on a data d and the user x triggered it. $WA^{s_i}_{Event}$ denotes the content of WA_{Event} at state s_i.

□

4.1.3 Security Mechanisms & Testing-Related Information

Users interact with a web application through a browser. Even if a user is not aware of what is happening, from a security perspective, a lot of information can be extracted from a web application about the mechanisms (that are often concealed to the users) implemented in order to preserve the security of the system (i.e., the security of the web application and its data-management technologies), and the information that is interesting from a testing perspective but is not part of the knowledge or the behavior of the web application.

Definition 4.6 (Assertions). Assertions describe security controls such as authentication, access control, input validation, encoding, and user and session management. A security analyst can define assertions (in the form of labels) about security controls in order to define the set *Assertion*.

□

The analysis is focused on those security mechanisms (enforced through/on some data) that refer to the classes in Definition 4.6. The strategy that a security analyst can follow when defining the set *Assertion* is to identify the

key attack surfaces that web applications can expose. This strategy corresponds to mapping the attack surface of the web application; some key areas to investigate during the mapping are:

- the web application's functionalities (i.e., the actions that can be leveraged);
- the core security mechanisms (e.g., access controls, authentication mechanisms, etc.);
- how the web application processes user-supplied input;
- the technologies employed on the client-side;
- the technologies employed on the server-side.

As an example, the following set is the one used in the case study:

$$Assertion = \{Granted, Checked, AJAX, Sanitized, Admin, User, Echoed, PageIncluded, noAttack\},$$

where the intended meanings for the elements in *Assertion* are:

- *Granted*: this label is used along with the data modeling the (HTTP) session and means that the session is granted (i.e., the user has logged-in and some session ID is used during the communication).
- *Checked*: if the data is used in a query on the database (or file system) and may not be displayed by the user interface.
- *Sanitized*: if sanitization is enforced on the data.[3]
- *Admin*/*User*: if the role of the users of the web application is checked (these assertions define the values of the user data *userType*, and are checked whenever an action requires these privileges).
- *AJAX*: if the data was displayed and its values are retrieved via AJAX-functionalities.
- *Echoed*: if the data submitted through a request is reported identical in the response page.
- *PageIncluded*: if in the URL/page there is a direct reference to a file (then used as a web page) hosted on the server.
- *noAttack*: this label is used as a means for a security analyst to disable attacks for a certain data.

Definition 4.7 ($SEC_{Assertion}$). The set

$$SEC_{Assertion} = \{\ (x, d, p)\ \mid\ x \in UserName, d \in Data \text{ and } p \in Assertion\ \}$$

states that a certain user x has used a data d on which the security analyst has made an assumption p about how the data d is handled from a security perspective; $SEC^{s_i}_{Assertion}$ denotes the content of the set $SEC_{Assertion}$ at state s_i.

□

[3] The only granularity considered is fully sanitized or not. The concretization phase will take care of testing the generated traces.

4.1.4 States of the transition system

In MobSTer, every state s_i describes a particular snapshot of the web application regarding the information about: (i) the users in *UserName* and their knowledge in $U^{s_i}_{Knows}$, (ii) the triggered events in *Event* on the data ($WA^{s_i}_{Event}$), and (iii) the assertions in *Assertion* about security mechanisms and testing-related information $SEC^{s_i}_{Assertion}$. In other words, a state describes what is happening client-side and server-side during the interaction of a user with the web application.

Definition 4.8 (States). For a given web application, a *state* of the *TS* is an instance of the matrix M such that the row names take values in *UserName*, the column names take values in every element of *Data* and the labels in K_{Source}, *Event* and *Assertion* are assigned to M's cells. The syntax $M[U, D]$ denotes a cell of the matrix M of the states of *TS*, where $U \in UserName$ and $D \in Data$.

□

A final remark on the matrix is in order. As stated above, MobSTer takes inspiration from the *HRU model* [39]. The definition of the access matrix (P in the original paper) remains the same but, in MobSTer , "subjects" are replaced by "users" and "objects" are replaced by "data" (the definitions of "commands" and "rights" are replaced in order to be usable in the context of web applications).

4.1.5 Actions

Let the following set be a set of labels for the modelled functionalities

$$functionName = \{Login, Logout, Search, GetEdit, ListId, \\ EditProfile, ViewProfile, UpdateProfile\}.$$

The elements in *functionName* refer to the functionalities implemented in the web application that are modeled as actions. These actions have to be instantiated with respect to the data (contained in the set *Data*) of the web application.

Definition 4.9 (Actions). An *action* $\alpha \in Action$ is defined as

$$\alpha = name(agent, parameters)/[Conditions]PrimitiveTransitions\ ,$$

where $name \in functionName$, $agent \in UserName$, $parameters \subseteq Data$, *Conditions* is a set of conditions that have to be satisfied in order to perform the action, and *PrimitiveTransitions* is a set of transitions that describe how the state changes.

□

Table 4.1: Definition of the *Login* action

```
1 Login(x, x.cred)
2  if (M[Anon, Anon.session] = Grant ∧ M[x, x.cred] = Initial)
3    Reset M for x
4    Del Grant from M[Anon, Anon.session]
5    Add Grant into M[x, x.session]
6    Add Checked into M[x, x.cred]
7 End
```

The elements of *PrimitiveTransitions* are of the form

```
operation X [into/from] M[U, D]
operation X for U
```

where $X \in Event \cup Assertion$, $U \in UserName$ and $D \in Data$. In the above example, the first primitive transition is applied to a single cell of M, whereas the second one is applied to all the cells in the row U.

A *condition* is an expression of the form:

```
if X ∈ / ∉ M[U, D] .
```

As an example, consider the definition of the action *Login*, which is given in Table 4.1. The action's *name* is "Login" (line 1) and, to maintain the functionality general enough to be used by multiple agents, let x be the agent using it. As *parameters* both "username, password" and "credential" could be used (instantiated for the user x). The parameter "credential" (*cred* for short) is used in the example in Table 4.1.

Usually, a login can be performed only if the user is not logged in yet (i.e., he is still anonymous to the web application) and if he knows his credentials (i.e., the credentials are part of his knowledge). These conditions are defined in line 2 of Table 4.1. Once the conditions in line 2 have been fulfilled, the state of the transition system needs to be changed. First of all, the previous event is deleted from the matrix (through the primitive `Reset` in line 3), then the *Grant* label is deleted from the *Anon* user in line 4 (this also means that a user can login only from an anonymous session) and the user receives the session in line 5. This information is also stored in the state (line 6) since the "credentials" are checked (the low-level mechanism is not important) and the action is closed in line 7.

4.1.6 Security goals

The purpose of security goals is to verify that some properties or conditions hold on the model. More specifically, MobSTer deals with injection flaws (SQLi, XSS and command injection).

In general, to exploit a vulnerability, some conditions related to the data and some properties must be fulfilled. Conditions are formalized with respect to a vulnerability by means of a logical formula that has to be valid in every

possible state describing the evolution of the model, or has to be valid for every trace starting from an initial state.

Although it is not possible to give an exact procedure for the definition of security goals, it is possible to give a general approach that can be helpful to a security analyst while writing security goals. The approach consists of four steps, which are described below. Each step should be instantiated according to the vulnerability for which the security analyst wants to write the security goal; examples of instantiations are given after each step.

1. *Definition of entry points:* The security analyst has to determine which are the entry points that are used to attack a web application. Examples of entry points are:
   - text/numerical parameters,
   - URLs,
   - login functionalities and
   - paths to files.
2. *Understanding the testing procedure:* The security analyst then has to understand which is the procedure used to test the vulnerability for which she wants to define a security goal (usually this procedure is the same as the one that is used to attack the web application). The OWASP testing guide [64] has an "How to Test" section for each covered vulnerability, and it is a good starting point for learning how to test web applications. For instance, the procedure for testing "SQL-Injections" described in [64] is:
   (i) Make a list of all input fields whose values could be used in crafting a SQL query.
   (ii) Test them separately, trying to interfere with the query and to generate an error.
   (iii) Monitor all the responses from the web server and have a look at the HTML/JavaScript source code for evidence of a successful attack.
   The procedure for testing "Directory traversal/file include" described in [64] is:
   (i) Enumerate all parts of the application that accept content from the user in order to load static information from a file.
   (ii) Insert malicious strings in the used parameter to include files that are not intended to be accessed by the user (e.g., "../../../../etc/passwd" to include the password hash file of a Linux/UNIX system). It is also possible to include files and scripts located on an external web site.
   (iii) Monitor all the responses from the web server and have a look at the HTML/JavaScript source code for evidence of a successful attack.
   The security analyst will gain two valuable items of information from the testing procedure. First, how the test should be performed, which will be included in the model of the web application as a goal defining how to find the entry point. Second, the set of payloads that should be used during the concretization phase.

3. *Model behavior association:* The security analyst has to define which behavior of the model is more suitable to describe the entry points and the state in which a web application should be after a successful test for the vulnerability. For example, the information that can be used for the definition of a login bypass via "SQL-Injection" is:
   (i) Username and password are submitted to the backend server.
   (ii) Username and password are checked against the information stored in the database.
   (iii) The restricted functionalities of the web application are usable.
4. *Logical formula definition:* The security analyst has to define a logical formula that merges all the information gathered during the previous steps with respect to the sets K_{Source}, *Event* and *Assertion* used to model the web application.

4.1.7 The Alloy language

Before showing the translation from the transition system to the Alloy language, I briefly introduce some basic notions of Alloy that are needed to understand the translation.

4.1.7.1 Signatures and Relations

A signature (`sig`) is the basic building block of the Alloy language and defines a set of elements. A signature can be specified to have always exactly one element by using the keyword `one`. For example,

```
one sig Obj {};
```

defines a set `Obj` that contains only one element. A signature can be defined to be `abstract` when one wishes to refine a classification of a set of elements. Each element contained in the abstract signature is constrained to also be contained in the signature that extends the abstract signature itself. An abstract signature can be extended by using the keyword `extend`. For example,

```
abstract sig Color {};

sig Blue extend Color {};

sig Red extend Color {};
```

defines an abstract signature `Color` and two signatures `Blue` and `Red` extending `Color`.

The body of a signature, which is contained within a pair of curly braces, allows one to define fields that declare relations between the set defined by the signature and another set or another relation. To define complex relations, Alloy provides multiplicity constraints and operators on sets. The multiplicity constraints relevant for MobSTer are:

- `x: set e`: meaning `x` is a subset of `e`;
- `x: one e`: meaning `x` is a singleton subset of `e`.

The operators on sets relevant for MobSTer are:

- `X + Y`: the union of sets `X` and `Y`,
- `X & Y`: the intersection of sets `X` and `Y`,
- `X - Y`: the difference of sets `X` and `Y`,
- `R -> S`: the product of two relations.

4.1.7.2 Facts

In Alloy, facts are used to put explicit constraints on the model. During the analysis of a model, any execution trace that violates any facts, will be discarded.

4.1.7.3 Predicates

Predicates allow one to specify parameterized constraints that can be used to represent operations. For example,

```
pred name [parameter1:domain1, parameter2:domain2]
     constraint1
     constraint2
     constraint3
```

If the inputs satisfy all of the specified constraints, then the predicate evaluates to true, otherwise it evaluates to false.

4.1.7.4 Assertion

Assertions are assumptions about the model that can be checked using the Alloy Analyzer. For example,

```
assert name-of-assertion
    // list of constraints
```

4.1.8 Model definitions in Alloy

It is now possible to define how to translate the transition system describing a web application in MobSTer into the Alloy language by means of the definitions in § 4.1.

4.1.8.1 Users, data and knowledge

The model of a web application in MobSTer can be formalized by applying Theorem 4.1, Theorem 4.2 and Theorem 4.3. The formalization of the model starts from the definition of the user's data structure (Theorem 4.1):

```
abstract sig User {

field1: one DataType1

field2: one DataType2

...

initialK: set Data

gainK: set Data

}
```

Translated in Alloy, the user is an abstract signature that contains a list of fields. Each user's field is also a signature that extends `Data`. The set `Data` represents the generic data type in the model, and the keyword `one` forces each field to contain exactly one element of the specified data type. In addition to the fields specifically related to the web application that is being modeled, the user's signature always defines the fields `initialK` and `gainK`. which are subsets of the set `Data` (denoted by the keyword `set`). Since `Data` is extended by all the other signatures of the user's data, `initialK` and `gainK` can contain any type of information. Two main data types are needed to model a web application in MobSTer:

- *basic*: defined by declaring a concrete signature that extends the abstract signature `Data`. For example, `sig BasicDataType extends Data`.
- *structured*: defined by declaring (i) an abstract signature A that extends the abstract signature `Data`, (ii) one or more fields in the signature A, and (iii) abstract signatures relative to the fields of the signature A. For example,

```
abstract sig StructDataType {
field1: one DataType1,
field2: one DataType2,
...
} extends Data
sig abstract DataType1 extends Data
sig abstract DataType2 extends Data
```

With the definition of the user and the data of the web application, it is possible to instantiate the user's fields.

- For each *basic* field, the concrete signature (one for each user) is defined as

```
one sig UserADataType1 extends DataType1 {}

one sig UserADataType2 extends DataType2 {}
```

- For each *structured* field, the concrete signature (one for each user) is defined as

```
one sig UserADataType3 extends DataType3{} {

field1 = UserADataType4

field2 = UserADataType5

}
```

- The concrete signature of each user and the relation associated to each field are defined as

```
one sig UserA extends User{}
{

field1 = UserADataType1

field3 = UserADataType1

field2 = UserADataType3

...

initialK = UserADataType1 + ...

gainK = NoData

}
```

4.2 Evaluation

To evaluate the effectiveness of MobSTer, two further steps were taken: (1) create a formal model on a web application following § 4.1 and (2) implement MobSTer in a prototype version that could be tested and compared with other tools.

To achieve (1), the Alloy language was selected for writing formal models of web applications so to be able to apply the Alloy analyzer [41] on such models. The Alloy analyzer is the tool used to analyze an Alloy model, it takes in input the model and its constraints and finds structures that satisfy them. Alloy has been used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks.

To achieve (2), a Python version of MobSTer performing a concretization phase was implemented. The implementation of MobSTer makes use of: the data contained in the CE (resulting from the model-checking phase), a

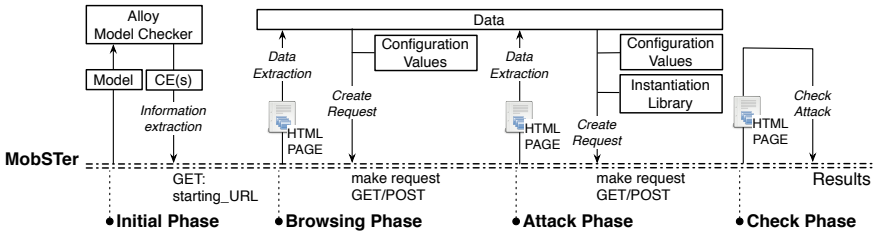


Fig. 4.2: Execution workflow of the MobSTer framework

file containing Concretization Values needed for the correct interaction with the SUT (e.g.: IP addresses, parameters names, etc.), and the Instantiation Library containing the attack-related information to perform the tests (i.e., payloads and scripts used during the actual attack against the SUT).

Listing 4.1: Structure of the configuration file used by MobSTer

```
1  starting_URL = 'http://192.168.1.42:8080/WebGoat/...'
2
3  set_cookie = {'JSESSIONID':'974
       AE36BC95C4I0D036C2E3CDS543B1C'}
4  SSL_authentication = [None|Yes]
5
6  views = {'Login':'ListId'}
7
8  Data['Tom'] = {
9   'employee_id' : '105',
10  'password'    : 'tom'
11 }
12 Data['Jerry'] = {
13  'employee_id' : '106',
14  'password'    : 'jerry'
15 }
16
17 actionsURL = {
18   'NoAction'  : '',
19   'Login'     : 'Login',
20   'Logout'    : 'Logout',
21    [...]
22   'GetSearch' : 'SearchStaff',
23   'Search'    : 'FindProfile'
24 }
25
26 actionsLabel = {
27   'NoAction'  : None,
28   'Login'     : None,
29   'Logout'    : None,
30    [...]
```

```
31    'GetSearch' : None,
32    'Search'    : None
33  }
```

4.2.1 Implementation

4.2.1.1 Initial Phase

In this phase, MobSTer automatically runs the Alloy Analyzer on the model of a SUT. If a CE is found during the initial phase, it is saved in a text file containing all the information about the states of the transition system. The text file is then parsed, along with the configuration file (containing the Concretization Values), in order to populate the Python variables that will be used in the next phases. Alloy supports multiple CEs generations, thus different test cases for the same security property on one model can be generated.

With the information regarding the CEs and the relative variables populated, the engine starts the interaction with the target web application. The security analyst is required to write a configuration file (Listing 4.1) for the target web application containing: (i) the URL from which to start the interaction (line 1 in Listing 4.1), (ii) two optional fields for setting a cookie and stating if the web application requires an SSL connection (lines 3-4 in Listing 4.1), and (iii) the list of concatenated actions (line 6 in Listing 4.1).

The notion of *views* (i.e., how the actions correspond to the real implementation of the web application with respect to its pages) is introduced in order to define the relations between the "pages" composing a web application and the functionalities provided by it (i.e., how they match). The definition of the views allows the security analyst to model each functionality as a single action without the need of defining actions referring to multiple functionalities. The Python engine is aware of these modeling choices via the use of the variable `views`.

4.2.1.2 Browsing Phase

After the initial phase, MobSTer has all the information required for the interaction with the SUT, thus the *browsing phase* can begin. This phase deals with the problem of reaching the exact location where the attack has to be performed. Knowing from the CE at which state the attack has to be made (say, at state 5), the Python engine selects, one by one, the actions from the first state to the one before the attack (in this example, state 4). For each action, the engine performs two sub-phases: "Data extraction" and "Request creation".

Data extraction: This phase assumes that a web page has been previously retrieved by the engine; for the very first interaction (i.e., the action `NoAction`) the engine retrieves the page pointed by the `starting_URL` variable). The

engine extracts and saves the data contained in the retrieved HTML page (e.g., text in input forms, checkboxes, etc.). For each user of the target web application, the data pertaining to the user is maintained in a data structure containing all the data gathered during the analysis (lines 8-15 in Listing 4.1).

Request creation: In order to create requests, the Python engine has to retrieve the data pertaining the request and generate the actual HTTP requests. The following sub-phases execute:

- *Data selection*: The engine checks all the data used in the model (i.e., the data type used during the modeling phase) in order to determine which one should be used (e.g., it tries to discriminate which `employee_id` and `password` should be used for the *Login* action); if multiple data can be selected in order to use an action, the engine automatically checks which data has to be used with a simple comparison with the data displayed, or used, in the subsequent state of the transition system, i.e., by tracking the evolution of the atomic propositions though the different states of the transition system
- *Input filling*: The engine selects the proper data from the one available in its data structure with respect to the possible input on the page (i.e., the possible HTTP requests that can be made). This is done, at first, by checking (i) if any of the labels (contained in a predefined list) is used in the page (e.g., *text, TEXT, password, textarea*), (ii) a list of indicators for buttons (e.g., *action, submit, button*) and (iii) which information is available in the extracted data (i.e., the data structure `Data`). In case multiple forms are present in a page, this information is not enough to decide which one should be used to create a HTTP request. To resolve this decisional problem, the security analyst specifies in the configuration file some information (associated with the action) that can be used for making such choices (e.g., the button action or the target page of a link; lines 18 and 27 in Listing 4.1). If the tool is not able to derive any of these information, the missing fields are reported to the security analyst with the request of supplying the missing value.
- *Request generation*: Once the correct data has been selected regarding a certain state of the transition system, the engine is ready to create and send a HTTP request to the SUT and, subsequently, to retrieve the resulting web page. The Python engine can generate various types of requests (i.e., GET/POST requests with variables, cookies or SSL authentication). To differentiate such cases, the security analyst should specify for each action a label (referring to the main feature modeled by the action) in the variable `actionsLabel` of the configuration values. The possible values (and their meaning) for `actionsLabel` are:
  - `none`: for buttons or forms (it is assumed forms to be one of the main features of web applications that are modeled);

- `link`: for the cases where a link has to be followed in order to reach a different part of the web application (e.g., to reload the information on a web page);
- `GP`: when the engine has to retrieve the given web page before accessing the functionality (`GP` stands for "Get Page").

4.2.1.3 Attack Phase

Once the browsing phase has been completed (i.e., the engine reached the location where the attack has to be made), the Python engine switches to the attack phase. During this phase, the engine delivers the payload for a given attack. The penetration testing approach for such a phase is to test every entry point on the target page, i.e., using every available input on the page in order to deliver every possible payload in the Instantiation Library. A complete scan of the web applications used as case studies (i.e., WebGoat, DVWA and Gruyere) is not the focus of this thesis (and their vulnerabilities are well known anyway); thus, in the implementation of this phase, the Python engine is given the knowledge of the entry points for each of the tested attacks (i.e., the right target field in a form or the partial URL to be used). This choice is not a limitation since the number of entry points can be increased if necessary and this information is only used during the attack phase (i.e., the other phases do not have any knowledge about the exact location of the vulnerabilities and everything is derived from the model and the interaction with the web application). In this way, it is possible to show the effectiveness of the Python engine with a reduced overhead during the attack phase. Of course, with an augmented number of entry points the overall time of the test will increase, but such analysis is not in the scope of this thesis.

4.2.1.4 Check Phase

After the delivery of every payload in the previous phase, a check phase starts. If the information contained in the CE requires that the check for an attack has to be made in a different location of the SUT (i.e., the success of the attack is not verifiable in the received HTTP response), an additional browsing phase is called. Once the engine retrieves the HTTP page where the results of the attack should appear, the engine checks if the attack was successful or not. Two types of checks are implemented in the prototype version of MobSTer: (i) *general checks* and (ii) *payload-related checks*.

General checks include those general conditions of the page that have to be checked for every payload of a given attack (e.g., search for a regular expression in the page, check a given status code, etc.). Payload-related checks include the checks for those attacks where the conditions that have to be checked depend on every single payload. Both *general checks* and *payload-related checks*, for the payloads considered by MobSTer, are included in the

Instantiation Library so that the security analyst does not have to manually define them. As an example, for the following payload for a XSS attack

```
alert(String.fromCharCode(88, 83, 83, 65,
    116, 116, 65, 99, 107))
```

the string XSSAttAck (i.e., the decoded string of the payload) has to be found in the result page in order to have the confidence that the attack was successful. Even though the checking phase implemented in the prototype version of MobSTer is simple, the modularity of the tool allows security analysts to implement and use the test oracle of their choosing.

4.2.2 Results of the tests

I now show the vulnerability coverage and effectiveness provided by MobSTer. In particular, MobSTer was tested against four security tools that are the closest to what MobSTer aims to achieve: Burp Suite [68] (free version 1.7.23 and Pro version 1.7.23), OWASP Zed Attack Proxy [62] (ZAP, version 2.3.1) and Arachni [5]. For performing the comparison all tools were configured in complete automatic mode with the default configuration that comes after the installation.

Arachni is a security scanner framework, whereas Burp Suite, ZAP and Paros are mainly proxy tools used to intercept and analyze the HTTP traffic from and to a SUT, but they also provide vulnerability-scanning techniques that we employed for the tests. The tools selected for testing are not the most powerful tools for performing vulnerability scanning, but still they are the main general-purpose and free tools currently available. The case studies used to perform the comparison are well known case studies able to show whether or not a class of vulnerability was being covered, specifically: WebGoat, Gruyere [44], Damn Vulnerable Web Application (DVWA [30]).

- WebGoat is a deliberately insecure web application developed and maintained by OWASP. WebGoat is divided in many different lessons which purpose is to teach how different vulnerabilities work;
- Gruyere is a small web application containing multiple security bugs that allows its users to publish snippets of text and store assorted files. The Gruyere model is composed of actions that permit users to add/delete messages (snippets), upload files, modify their profile, and see other user profiles;
- DVWA is a PHP/MySQL web application that, similarly to WebGoat, is divided in different lessons and shows different web application vulnerabilities. DVWA allows the selection of three levels of security to test, making each lesson harder to exploit.

Table 4.2 shows a tabular representation of the results of the tests and in the following I describe in detail the results dividing them by category of attack following the categorization in the table.

Table 4.2: Results of the tests performed on the case studies. Legenda: ✓ – success of the test, i.e., the test has been able to find the vulnerability, X – failure of the test, ∼ – ineffective test or the test is inapplicable to the case study

	Case Study	MobSTer	Burp	Burp Pro	ZAP	Arachni
Access Control Flaws						
Business Layer Access Control	WebGoat	X	∼	∼	∼	X
AJAX Security						
DOM-Injection	WebGoat	✓	X	X	∼	X
Reflected XSS via AJAX	Gruyere	✓	X	✓	X	X
Cross-Site Scripting (XSS)						
Stored XSS	WebGoat	✓	X	X	X	X
	Gruyere	✓	∼	X	X	X
	Gruyere	✓	∼	X	X	X
	DVWA	✓	✓	✓	✓	✓
Reflected XSS	WebGoat	✓	✓	✓	✓	∼
	WebGoat	✓	✓	✓	X	∼
	Gruyere	✓	∼	✓	∼	∼
	DVWA	✓	✓	✓	✓	✓
Injection Flaws						
Command-Injection (or Execution)	WebGoat	X	X	X	X	X
	DVWA	✓	✓	✓	✓	✓
SQL-Injection (string, numeric, XPATH)	WebGoat	✓	✓	✓	✓	∼
	WebGoat	✓	✓	✓	✓	∼
	WebGoat	✓	✓	✓	X	∼
	WebGoat	✓	✓	✓	✓	∼
	DVWA	✓	✓	✓	✓	✓
	DVWA	✓	✓	✓	✓	✓

4.2.2.1 Access-control flaws

Access-control flaws violate business-level policies and they are extremely difficult to characterize and identify automatically since those policies are unique for each SUT. In the case studies, used to evaluate MobSTer, access control policies were defined as goals of the model. The goals check if the user has the correct privileges when accessing an action.

- **Business layer access control**: In this lesson, MobSTer was able to gain access to high-privilege actions with a low-privilege account. The vulnerability was tested but a false negative reported (i.e., the vulnerability was exploited but not reported as such). The main reason for MobSTer to be able to exploit this type of vulnerability is that with a model of the SUT it is possible to represent the means to reason about roles (as abstract

security constraints) and derive traces that are meaningful for these tests. However, the fact that the vulnerability was exploited but not reported, highlights the fact that MobSTer requires a more refined attacks check phase.
On the other hand, the benchmark tools focus their security evaluation on the analysis of the raw HTTP messages exchanged between the browser and the SUT and do not have knowledge of the meaning of the data in these messages.

4.2.2.2 AJAX security

To detect and test this type of flaws, the goal checks if it is possible to perform an attack with the use of actions that employ AJAX technologies.

- **DOM-Injection**: MobSTer was able to perform tests for blocked functionalities and bypass the restrictions coded in the DOM. The other tools failed to identify this attack since they could not perform the request containing the vulnerability. The model that was created for this type of attack provides an example of how different types of functionalities can be modeled in MobSTer and how a security analyst can leverage information that other security tools cannot see.
- **Reflected XSS via AJAX**: MobSTer was able to find and confirm the attack thanks to its concretization methodology. Specifically, being able to control where the attack has to be launched makes the testing for a distinct vulnerability stronger than a broad search for many vulnerabilities. Only Burp Pro was able to perform the attack thanks to its scanner engine.

4.2.2.3 Cross-Site Scripting (XSS)

The considered XSS flaws are:

- **Stored XSS**: The goal checks if there exists an action where a data, previously written by a user, is displayed by a (different) user. MobSTer was able to trigger the XSS attacks in locations different from the ones used to deliver the payload. More specifically:
  - In the WebGoat and Gruyere case studies, the success of MobSTer resides in the attack traces generated by the model checker. Usually, this type of vulnerability is not found with automatic tools since it requires one to check if a specific payload is executed in a different location where it was injected. MobSTer presents a clear advantage in finding and exploiting this vulnerability since it allows for this possibility.
  - In the DVWA case study, the stored XSS behaves very similarly as a reflected XSS since the payload, once stored in the database, is also sent back in the response from the SUT. Because of this characteristic , all the tools were able to find the vulnerability.

- **Reflected XSS**: The goal checks if there exists an action where the user input is displayed back to the same user. MobSTer was able to find the vulnerabilities. Like the other examples, this attack is well known and modern security tools are using mature testing methodologies. For the tests performed with Burp free it was necessary to perform a manual check since Burp free does not automatically check if a payload is present in the response. Burp Pro, on the other hand, provides an automatic checking phase that was able to identify the vulnerability. For the tests performed with Arachni, it should be noted that it was not possible to aim correctly the tool on the vulnerable page since WebGoat loads the content of a page based on the value of some parameters in the HTTP request's body, which Arachni does not allow one to specify. The only other way to make body parameters visible to Arachni is to use its integrated proxy feature. We tried it and although Arachni was now able to test the body parameters, the set of payloads used by Arachni caused WebGoat to generate numerous errors, which resulted in Arachni not being able to find the vulnerability.

4.2.2.4 Injection flaws

The goals for the injection flaws check if a user can access an action that was previously injected by an attacker.

- **Command-Injection (or execution)**: The goal for this vulnerability checks if there exists an action where user supplied input is used as part of the execution of an operating system command.
  - In the case of WebGoat, MobSTer was able to find the vulnerability during the model-checking phase but the TEE could not exploit it. Also Burp (Free and Pro), Arachni and ZAP failed to find the vulnerability. The main problem in this example was that the standard security policies of the Tomcat server blocked the execution of shell commands.
  - In the case of DVWA, MobSTer, Burp (Free and Pro), ZAP and Arachni were able to successfully complete the tests (i.e., find the vulnerability).
- **SQL-Injection attacks**: The goal for this vulnerability checks if, during an action, external input was used in a query on the database resulting in additional data displayed by the user interface. MobSTer was able to perform the tests with success. Burp (Free and Pro) also proven to be successful in performing these tests; this is due to the fact that the payloads used for the tests are the same even though a manual check of the success of the tests is required with Burp free. For Arachni, the same problem that occurred with the "Reflected XSS" case studies occurred: the payloads generated by Arachni caused numerous crashes on WebGoat.

4.3 Related work

Model-based testing approaches are the more closely related to MobSTer and thus call for a comparison with existing researches.

In [20], the authors describe an approach called "Chained Attack" that takes HTTP requests in input, generates a model, and searches for sequences of attacks on the model by exploiting model-checking techniques that implement the Dolev-Yao [27] intruder. The approach is similar to MobSTer, but the result they present is quite limited. Even though the approach aims to be used by security analysts that do not have a strong background in formal methods, quite a lot is required of them for the concretization phase. Security analysts have to write a configuration file containing information such as header names and keys appearing in raw HTTP messages, semantics of HTTP responses and conditions representing attacks, whereas MobSTer simplifies the definition of the models and provides more automation for the concretization phase. More specifically, their concretization phase is not as mature as MobSTer's since they provide only information about the payloads that need to be used, whereas MobSTer provides for a more solid testing engine with its Instantiation Library.

The methodology proposed in [7, 9] is focused on binding the specification of security protocols to actual implementations. The methodology starts with the definition of an abstract model (written using a role-based language) of the HTTP messages composing the security protocol. A model checker is then used in order to derive a counterexample violating some given security properties. The abstract messages, contained in the counterexample, are then mapped to concrete messages used to test the web application. The results are particularly promising but not directly comparable to MobSTer, since MobSTer is at a different level of abstraction. Indeed, the models MobSTer deals with are not protocol-dependent (HTTP messages) but application-dependent (actions).

The work described in [17, 18, 19] presents an approach for model-based security testing of web applications closely related to mutation testing: they start from a secure model and use mutation operators to automatically introduce vulnerabilities in the model. Some of the major differences with MobSTer reside in modeling the interaction between user and web application (MobSTer takes into consideration the user interaction only when the CEs are tested), the use of mutations and the use of an attacker in the model-checking phase. On the other hand, the approach by [17, 18, 19] is more mature than MobSTer's and allows for automatic test-case generation via mutants.

The work in [3] proposes a methodology that relies on the Alloy Analyzer for the analysis of several sample web mechanisms and web applications. The authors employ threat models such as a malicious attacker controlling a website or even a portion of the network. They have defined three different intruder models that should find web attacks, whereas in MobSTer it is only required to describe the behavior of the web application. Furthermore, this method-

ology is at a different level of abstraction than MobSTer since they mainly model network infrastructures and protocols rather than web applications.

The work in [45] describes an approach for the identification of vulnerabilities based on the formalization of vulnerability test patterns into test purposes. They define both the behavior of the web application and the test purpose, and use model-checking techniques for the generation of abstract test cases. The idea is similar to MobSTer, but MobSTer allows for a wider coverage of vulnerabilities, whereas [45] considers the main vulnerabilities that are provided in terms of test patterns from CWE (i.e., Blind and Not Blind SQL-Injection, Reflected and Stored XSS) and that need to be translated in order to be tested. The coverage of MobSTer is wider as it considers also more complex attacks that exploit logical flaws. [82] extend their work ([45]) with risk assessment in order to select relevant aspects and vulnerabilities for the generation of models. The same differences with respect to MobSTer pointed out for [45] apply for [82].

In [16], the authors propose a model-based vulnerability testing approach where attacker models are used to test a given web application (or functionality). In this approach, the payloads and the behavior of the attacker are separated. Attacker models can be seen as an instantiation of the security goals that are defined in this work, although the level of abstraction of the security goals described in this work is higher (in the sense that the attacker models are specific to a scenario) than the one proposed in [16]. The main difference with respect to this approach is that MobSTer bases the analysis on the models of web applications rather than the procedures for testing them; this results in not needing to change such procedures if a (slightly) different web application has to be tested.

Another formal approach that uses a different attacker model is proposed by [70] where they use the formal language ASLan++ [85] and model-checking techniques that implement the Dolev-Yao [27] intruder model for modeling a web application vulnerable to CSRF but they do not consider other vulnerabilities.

4.4 Conclusions

In this chapter I have described MobSTer, a framework that was developed during the doctoral thesis of Dr. Michele Peroli in which I collaborated at the beginning of my PhD. MobSTer combines model-checking techniques and penetration testing check lists to automate the testing of web applications without missing important checks. Models of web applications are created by taking into consideration different aspects, in particular: (i) their functionalities (that are modeled as actions), (ii) the data used (along with information about data storage and management), and (iii) a security goal specifying the vulnerability to be tested.

As became clear during the testing phase, whose results are shown in the evaluation section (§ 4.2), the use of actions has a positive impact on all the phases of MobSTer, resulting in a quite simple and flexible methodology for testing the security of web applications. Finally, one strength of great impact provided by MobSTer is the reusability of actions and of the sets K_{Source}, *Event* and *Assertion*. The expertise required to populate the Instantiation Library is automatically "reused". The analyst can collect her expertise into MobSTer and reuse it during future tests on possibly different web applications, which may be carried out by her or by members of the testing group of the analyst's organization, if any. New attack techniques and payloads can be inserted into the framework without changing (or compromising) the testing methodology.

Part III

Multi-stage analysis of web applications

5

The formalization

I now describe how to formally represent the behavior of a vulnerable web application and how the Dolev-Yao (DY) attacker model can successfully exploit vulnerabilities of web applications. In my formalization, I use ASLan++, the formal specification language of the AVANTSSAR platform [6] and the SPaCIoS Tool [84], but in fact the approach that I propose is general and, mutatis mutandis, it could be quite straightforwardly used with other specification languages and/or other reasoners implementing the DY attacker model.

Before providing the details of the formalization, I highlight an important point about this work: WAFEx does not search for payloads that can be used to exploit vulnerabilities, but rather it analyzes the security of web applications by exploiting multiple vulnerabilities that lead attackers to violate a security property of a web application. This approach can successfully exploit and combine vulnerabilities related to:

- SQL injection,
- File-system access,
- Cross-Site Scripting and
- Cross-Site Request Forgery,

but, again, the approach that I propose is general enough and it can be, fairly easily, extended to cover more classes of vulnerabilities. My formalization represents five main entities:

(i) the *web attacker*, which interacts with the web application and the honest client,
(ii) the *file-system*, which interacts with the web application and the database for reading and writing content,
(iii) the *database*, which interacts with the web application and the file-system, providing a means to query a database,
(iv) the *web application*, which defines the interaction with the web attacker, the file-system, the database and the honest client,

(v) *honest client*, which interacts with the web application and the web attacker,

The entities *web attacker*, *file-system*, *database* and *honest client* can be reused in every formalization of a web application, whereas the entity *web application* should be specified according to the web application under analysis. In the following, I first define the data types and the communication model adopted in the formalization, and then I describe the main entities of the formalization.

5.1 Data types

As already mentioned in § 3.1.2.2, ASLan++ data types can have sub-types. This is particularly useful to keep the analysis simple and efficient. I now define the sub-types used in the formalization. Recall that in ASLan++ sub-types are represented as a relation `subtype < message` meaning that any value of type `subtype` can be used in a context where a value of type `message` is required, but a value of type `message` cannot be used in the context where a value of type `subtype` is required. Listing 5.1 defines the sub-types `cookie` (representing cookies), `param` (representing HTTP parameters), `inode` (representing a file in the file-system) and `nonce` (representing a fresh value) to be subtypes of `text`, and `page` (representing a web page) to be sub-type of `inode`.

Listing 5.1: ASLan++ code of the sub-types used in my formalization

```
types
 cookie < text;
 param < text;
 inode < text;
 nonce < text;
 page < inode;
```

5.2 The communication model

Messages are the basic communication units used in my formalization and are defined as follows:

Definition 5.1. Messages *consist of* variables V, constants c, concatenation $M.M$, function application $f(M)$ *of uninterpreted function symbols* f *to messages* M, *and* encryption $\{M\}_M$ *of messages with public, private or symmetric keys that are themselves messages.*[1] M_1 *is a* submessage *of* M_2 *as is standard (e.g.,* M_1 *is a submessage of* $M_1.M_3$, *of* $f(M_1)$ *and of* $\{M_1\}_{M_4}$*) and, abusing notation, I then write* $M_1 \in M_2$.

[1] I need not distinguish between different kinds of encrypted messages, but it could be possible to do it by following standard practice.

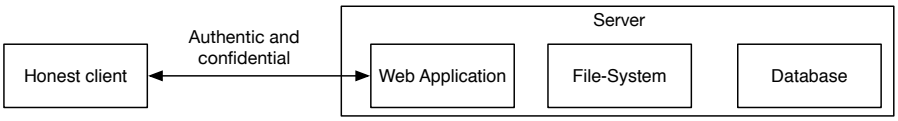


Fig. 5.1: The communication model between the honest client, the web application, the file-system and the database

□

As illustrated in (Figure 5.1), for the communication model of my formalization, I assume:

1. that the honest client and the web application communicate over an authentic and confidential channel, and
2. that the web application, the file-system and the database are on the same physical machine i.e., no attacker can read or modify the communication between them.

These assumptions derive from the fact that I am not interested in finding attacks against the communication between honest client and web application but rather to exploit vulnerabilities of the web application — this is not a limitation as the formalization could be quite straightforwardly extended to include attacks to the communication channel. Furthermore, the file-system and the database entities are not real network nodes and thus no attacker can put himself between the communication, i.e., man-in-the-middle attacks are not possible.

In order to express this assumption in ASLan++, I formalize the communication between the honest client and web application by using a confidential and authentic channel, represented in ASLan++ by means of the `*->*` notation. Moreover, it is needed to ensure a correct pairing between requests and responses. This is done by including a fresh nonce in every request.

The ASLan++ code that formalizes this communication model is given in Listing 5.2, where two entities are involved in the communication: a client (line 1) and a server (line 12).

Listing 5.2: ASLan++ code implementing a communication model enforcing authenticity, confidentiality and freshness

```
1  entity Client (Actor, S : agent) {
2   symbols
3    Nonce : text;
4    payload, Response : message;
5   body{
6    Nonce := fresh();
7    Actor *->* S : payload.Nonce;
8    S *->* Actor : ?Response.Nonce;
9   }
```

```
10 }
11
12 entity Server (Actor, C : agent) {
13  symbols
14   NonceC : text;
15   response, PayloadC: message;
16  body{
17   while(true){
18    select{on(S *->* Actor : ?PayloadC.?NonceC):{
19      Actor *->* S : Response.NonceC;
20     }}
21   }
22  }
23 }
```

The client defines a variable `Nonce` to use as nonce (line 3), a constant `payload` to use as message to send to the server (line 4) and a variable `Response` used to store the response from the server (line 4). The body of the client starts by initializing the variable `Nonce` to a fresh value (line 6). The client sends a message to the server by concatenating the constant `payload` and `Nonce` (line 7), and then expects to receive a response from the server containing the value for the variable `Response` and the same nonce sent to the server in the previous request (line 8).

The server defines a variable `NonceC` used to store the nonce received from the client (line 14), a constant `response` representing the response the server sends to the client and a variable `PayloadC` used for receiving the payload sent from the client (line 15). The server is a network node that actively listens for incoming connections, thus its body is enclosed in an endless while loop (line 17). In case it receives a message from the client (line 18), it stores the value of the received message in `PayloadC` and the value of the nonce in `NonceC`. Finally, the server responds to the client with the constant `response` and the nonce sent from the client (line 19).

The communication between web application, file-system and database does not involve the exchange of any message (as I assumed they are on the same physical machine). The communication between these entities is thus represented by means of facts and facts retraction. Every operation that can be performed on either the database or the file-system is associated to a fact. The web application uses a fact associated to an action (e.g., perform a query, read a file) and, once the operation is completed, the entity entitled to execute that operation (the database or the file-system) uses fact retraction to remove the fact and restore execution to the web application.

Listing 5.3: ASLan++ code implementing the communication model used between web application, database and file-system

```
1 entity Env{
2  symbols
3   nonpublic query(message set, message) : fact;
```

```
4    Result : message;
5
6   entity Server {
7    symbols
8     sql : message;
9     table : message set;
10   body{
11    query(table, sql);
12    select{on(!query(table, sql)):{
13      % handle variable Result here
14    }}
15   }
16  }
17
18  entity Database {
19   symbols
20    Sql: message;
21    Table : message set;
22   body{
23    while(true){
24     select{on(query(?Table, ?Sql)):{
25       Result := <...>;
26       retract query(Table, Sql);
27      }}
28    }
29   }
30  }
31  ...
32 }
```

Listing 5.3 shows an example of such a communication model. Within the `Env` entity I define a nonpublic fact `query(message set, message)` used to represent the execution of a query, and a variable `Result` that is shared between all the subentities of `Env`: `Server` and `Database`. The shared variable is used to store the result of executing the query (lines 3 and 4). The entity `Server` (line 6) defines symbols needed to execute the query: the query itself (line 8) and the table over which the query should be executed (line 9). The fact `query(table, sql)` is then used for executing a query (line 11). The server then waits for the fact to be retracted (line 12), meaning that the query has been executed. Finally, the server finds the result of executing the query in the shared variable `Result` (line 13).

In the database (line 18), variables are defined to represent the table (line 20) and the query (line 21). The database then actively waits to process new queries within and endless while loop (line 23) and when a query fact is valid (line 24) the value of the shared variable `Result` is initialized accordingly to the behavior of the query being executed (line 25). Finally, the fact representing the query is retracted (line 26) so that the entity that requested the execution of the query can continue its execution.

5.3 The Web Attacker

The web attacker that I propose can be reused in every specification and is based on the canonical model by Dolev and Yao [27], which defines an attacker that has total control over the network but cannot break cryptography: he can intercept messages and decrypt them if he holds the corresponding keys for decryption, and he can generate new messages from his knowledge and send messages under any agent name. Message generation and analysis are formalized by derivation rules, expressing how the attacker can derive messages from his knowledge. The web attacker I propose originates from two fundamental aspects of my work: (1) as stated, I am not interested in generating the payloads that will exploit vulnerabilities, but rather I want to represent that a vulnerability can be exploited and what happens when it is exploited, and (2) the models should be as simple as possible so as to avoid state-space explosion when carrying out the analysis with the model checker.

With these two aspects in mind, suppose that an attacker wants to search for a possible SQLi attack in a login form possibly leading him to have unauthorized access to the web application. As described in § 2.1, the attacker injects a statement that changes the truth value of a `WHERE` clause in a SQL `SELECT` query, creating a tautology. Similarly, suppose the attacker is trying to look for a file inclusion vulnerability possibly leading him to access resources stored outside the root folder of the web application (Directory Traversal vulnerability). As described in § 2.2, the attacker tries to access resources by injecting an appropriate payload that forces the web application to read files outside of the root directory of the web application. Generally, whenever the attacker wants to exploit one of the vulnerabilities described in Chapter 2, it has to inject an appropriate malicious payload that forces the web application into behaving in a way different.

In WAFEx I do not explicitly represent payloads since finding the right payload is not in the scope of this work; I thus introduce a constant `malicious` that represents any and all malicious inputs the attacker could use for exploiting one of the vulnerabilities described in Chapter 2. In the case of a SQLi attack, the constant `malicious` represents the attempt to create a Boolean tautology such as `' or '1'='1`, whereas in the case of a file inclusion vulnerability it represents the attempt to access a file outside of the root directory of the web application such as `../../etc/passwd`. To represent an attacker's attempt to exploit a vulnerability, I define the following Horn clause

```
hc_evil(M) : attack(M) :- M=malicious.?;
```

that states that the predicate `attack(M)` holds for a message `M` whenever it is of the form `malicious.?`, i.e., a message `M` that is a concatenation of any message (represented by `?`) and the constant `malicious` that represents a payload to exploit web applications vulnerabilities. More specifically, this states that the attacker has injected a malicious payload `malicious` into the parameter (expressed as a variable) `M`. In case of a SQLi attack for authentication

bypass, one can think of `malicious` as the `' or '1'='1` payload that creates a tautology allowing unauthorized access to the web application.

My approach allows the DY attacker to exploit the following attacks:

- SQLi for extracting the content of the database,
- SQLi for creating a tautology,
- SQLi for reading from the file-system,
- SQLi for writing to the file-system,
- SQLi for inserting new content into the database,
- file inclusion for reading arbitrary content from the file-system,
- remote code execution by uploading an arbitrary malicious file,
- XSS (stored and reflected) for redirecting a user,
- XSS (stored and reflected) for stealing the user's knowledge,
- CSRF for forcing the user into performing a request chosen by the attacker.

To be able to represent these attacks, the entities `file-system`, `database` and `honest` formalize their behavior based on the validity of the Horn clause `attack` which means they define how they should respond in case they receive a message containing `malicious`. By doing so, the DY attacker can exploit one of the vulnerabilities previously described by simply generating a message that contains `malicious`.

5.4 The File-system

I now give the formalization of the file-system entity that can be used in any specification when searching for attacks related to file-system vulnerabilities of web applications. The file-system is always actively listening for incoming requests from the web application and database. As already stated, I assume the file-system to be on the same physical machine of the web application (Figure 5.1), thus no attacker can put himself between the communication (i.e., man-in-the-middle attacks are not possible).

My formalization aims to abstract as many concrete details as possible, while still being able to represent the exploitation of file-system vulnerabilities, and so I do not represent the entire file-system structure but rather formalize messages sent and received along with reading and writing behavior. This allows to give a compact formalization so as to avoid state-space explosion problems when carrying out the analysis with the model checker. The ASLan++ code representing the file-system entity is given in Listing 5.4.

Listing 5.4: ASLan++ code representing the behavior of the file-system entity

```
1 nonpublic fs : message set;
2 nonpublic file(message) : message;
3 nonpublic readFile(message) : fact;
4 nonpublic writeFile(message) : fact;
```

```
5  ...
6  entity Filesystem {
7   symbols
8    Path, Val : inode;
9   body{
10   while(true){
11    select{
12     % file-system read
13     on(readFile(?Path)):{
14      select{on(fs->contains(file(Path))):{
15       Result := file(Path);
16       retract readFile(Path);
17     }}}
18     % file-system write
19     on(writeFile(?Path)):{
20      fs->add(file(Path));
21      retract writeFile(Path);
22     }
23     % file-system malicious read
24     on(readFile(?Path)):{
25      select{on(attack(Path)):{
26       select{on(fs->contains(file(?Path))):{
27       Result := Path;
28       retract readFile(Path);
29     }}}}}
30    }
31   }
32  }
33 }
```

5.4.1 File-system content

To exploit vulnerabilities related to reading and writing operations, I represent the files available in the file-system. The existence of a file is represented by means of a set of messages `fs` (line 1) and the uninterpreted function `file` (line 2), i.e., given a variable `Filepath` of type `message`, `file(Filepath)` represents a file in the file-system and `fs->contains(file(Filepath))` is used to verify whether or not the file `file(Filepath)` is stored in the file-system.

5.4.2 File-system operations

The file-system entity reacts to reading and writing operations, for which I define the facts `readFile(message)` (line 3) and `writeFile(message)` (line 4), both taking a generic variable of type `message` that represents a file location in the file-system.

5.4.3 Reading and writing behavior

The file-system entity formalizes a generic file-system so that this entity can be reused in every scenario where a web application interacts with a file-system. Before detailing the formalization of reading and writing behaviors, I recall that the goal of my approach is to test the security of a web application and thus access control policies/models for the file-system are not considered as they are external to the web application. Hence, I assume that every file that is in the file-system can always be read and that every writing operation will always succeed.

To define reading and writing behavior, the file-system entity defines, within the symbols definition (line 7), two variables `Path` and `Val` of type `inode` that are used to identify two different files (line 8). The file-system entity is represented as an entity always ready to execute new operations (line 10). More specifically, I define the file-system with three `select-on` branches representing the behaviors of: a reading operation (line 13), a writing operation (line 19) and a malicious reading operation (line 24).[2]

Whenever the fact `readFile(Path)` holds (line 13), the file-system entity checks, by means of the `fs` set, whether the file exists in the file-system (line 14): if so, the file-system stores `file(Path)` in the variable (shared with the web application) `Result` (line 15) and retracts the fact `readFile(Path)` to notify that the reading operation has been performed (line 16).

Whenever the fact `writeFile(Path)` holds (line 19), the file-system entity adds the existence of the file `Path` to the set `fs`, i.e., `fs->add(file(Path))` (line 20). As result of a writing request, the file-system entity need not return a result as I do not consider access control policies and thus I assume that a writing operation will always succeed. The file-system retracts the fact `writeFile(Path)` (line 21) to notify that the writing operation has been performed.

Finally, the file-system entity specifies what happens in the case of a malicious attempt from the attacker to access the file-system for a reading operation (line 24). In this case, whenever both the fact `readFile(Path)` and the Horn clause `attack(Path)` hold, the attacker has the ability to access an arbitrary file stored in the file-system. To represent this behavior, the file-system makes use of the non-deterministic nature of the `select-on` statement[3] in order to retrieve an arbitrary file from the file-system (line 26) and stores it into the shared variable `Result` (line 27). Finally, the file-system retracts the fact `readFile(Path)` in order to notify that the reading operation has been performed (line 28). During the analysis, the model checker will determine the appropriate file that should be retrieved in order to violate the security property defined on the model.

[2] I need not define a malicious writing operation since I assumed a writing operation to always succeed due to not considering access control policies.

[3] See § 3.1.2.2 for further details on the non-deterministic nature of the `select-on` statement in ASLan++.

5.5 The Database

I now give the formalization of the database entity that can be used in any specification when searching for SQLi attacks in web applications. The database, just like the file-system, is always actively listening for connections. I assume the communication between the database and the web application and between the database and the file-system to be secure since the database actually is not a real network node, and thus no network attacker can put himself between the communication with the database, i.e., man-in-the-middle attacks are not possible.

I propose a formalization that can be used in any specification and that is both *compact*, to avoid state-space explosion problems, and *general enough* not to be tailored to a given technology (e.g., MySQL or PostgreSQL). Hence, I do not represent the entire database structure, the SQL syntax nor access policies specified by the database. Rather, I formalize messages sent and received, the tables of the database and queries. The ASLan++ code representing the database behavior is given in Listing 5.5.

Listing 5.5: ASLan++ code representing the behavior of the Database entity

```
1  nonpublic query(message set, message) : fact;
2  nonpublic inset(message set, message) : fact;
3  nonpublic delete(message set, message) : fact;
4  nonpublic update(messageset, message, message) : fact;
5  nonpublic query_read(message) : fact;
6  nonpublic query_write(message) : fact;
7  ...
8  entity Database {
9   symbols
10   Table : message set;
11   Tuple, Sql, Oldtuple : message;
12   File : fnode;
13
14  body{
15    while(true){
16     select{
17     % query behavior
18     on(query(?Table, ?Tuple)):{
19      select{
20       on(!attack(Tuple) & (Table->contains(Tuple)):{
21        Result := Tuple;
22       }
23       on(attack(Tuple) & Table->contains(?Sql)):{
24        Result := Sql;
25       }
26       on(attack(Tuple) & fs->contains(file(?File))):{
27        Result := file(File);
28       }
```

```
29          on(attack(Tuple) & Tuple = malicious.?File.?):{
30           fs->add(file(File));
31           Result := file(File);
32          }
33          on(attack(Tuple)):{
34           Table->add(Tuple);
35           Result := Tuple;
36          }
37          on(attack(Tuple)):{
38           Result := db;
39          }
40         }
41         retract query(Table, Tuple);
42        }
43        % query insert
44        on(insert(?Table, ?Tuple)):{
45         select{
46          on(true):{
47           Table->add(Tuple);
48           Result := Tuple;
49          }
50          on(attack(Tuple) & Table->contains(?Sql)):{
51           Result := Sql;
52          }
53          on(attack(Tuple) & fs->contains(file(?File))):{
54           Result := file(File);
55          }
56          on(attack(Tuple) & Tuple = malicious.?File.?):{
57           fs->add(file(File));
58          }
59          on(attack(Tuple)):{
60           Table->add(Tuple);
61          }
62          on(attack(Tuple)):{
63           Result := db;
64          }
65         }
66         retract insert(Table, Tuple);
67        }
68        % query delete
69        on(delete(?Table, ?Tuple)):{
70         select{
71          on(true):{
72           Table->remove(Tuple)
73           Result := Tuple;
74          }
75          on(attack(Tuple) & Table->contains(?Sql)):{
76           Result := Sql;
77          }
```

```
78        on(attack(Tuple) & fs->contains(file(?File))):{
79         Result := file(File);
80        }
81        on(attack(Tuple) & Tuple = malicious.?File.?):{
82         fs->add(file(File));
83        }
84        on(attack(Tuple)):{
85         Table->add(Tuple);
86        }
87        on(attack(Tuple)):{
88         Result := db;
89        }
90       }
91       retract update(Table, Key, Tuple);
92      }
93      % query update
94      on(update(?Table, ?Oldtuple, ?Tuple)):{
95       select{
96        on(true):{
97         Table->remove(Oldtuple)
98         Table->add(Tuple);
99         Result := Tuple;
100       }
101       on(attack(Tuple) & Table->contains(?Sql)):{
102        Result := Sql;
103       }
104       on(attack(Tuple) & fs->contains(file(?File))):{
105        Result := file(File);
106       }
107       on(attack(Tuple) & Tuple = malicious.?File.?):{
108        fs->add(file(File));
109        Result := file(File);
110       }
111       on(attack(Tuple)):{
112        Table->add(Tuple);
113        Result := Tuple;
114       }
115       on(attack(Tuple)):{
116        Result := db;
117       }
118      }
119      retract update(Table, Oldtuple, Tuple);
120     }
121     % query read file
122     on(query_read(?Tuple)):{
123      select{
124       on(fs->contains(file(Tuple))):{
125        Result := file(Tuple);
126       }
```

```
127         on(attack(Tuple) & Table->contains(?Sql)):{
128          Result := Sql;
129         }
130         on(attack(Tuple) & fs->contains(file(?File))):{
131          Result := file(File);
132          retract query(Table, Tuple);
133         }
134         on(attack(Tuple) & Tuple = malicious.?File.?):{
135          fs->add(file(File));
136          Result := file(File);
137         }
138         on(attack(Tuple)):{
139          Table->add(Tuple);
140          Reulst := Tuple;
141         }
142         on(attack(Tuple)):{
143          Result := db;
144         }
145        }
146        retract query(Table, Tuple);
147       }
148       % query write file
149       on(query_write(?Tuple)):{
150        select{
151         on(true):{
152          fs->add(file(Tuple));
153          Result := file(Tuple);
154         }
155         on(attack(Tuple) & Table->contains(?Sql)):{
156          Result := Sql;
157         }
158         on(attack(Tuple) & fs->contains(file(?File))):{
159          Result := file(File);
160         }
161         on(attack(Tuple) & Tuple = malicious.?File.?):{
162          fs->add(file(File));
163          Tuple := file(File);
164         }
165         on(attack(Tuple)):{
166          Table->add(Tuple);
167          Result := Tuple;
168         }
169         on(attack(Tuple)):{
170          Result := db;
171         }
172        }
173        retract query_write(Tuple);
174       }
175      }
```

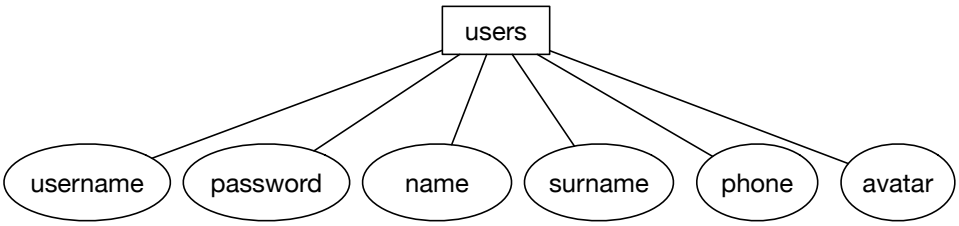


Fig. 5.2: Example of an entity-relationship model composed of one table and six attributes

```
176   }
177  }
178  }
```

5.5.1 Database content

As already stated, I want to avoid state-space explosion problems while being able to deal with the vulnerabilities described in §2.1. To do so, the database is defined as a set of sets, represented by the constant `db`, which contains one set for each table of the real database. The content of a table is then represented by one or more tuple defined as a concatenation of messages.

As an example, consider the simple entity-relationship model shown in Figure 5.2 where there is a table `users` containing six attributes: `username`, `password`, `name`, `surname`, `phone`, `avatar`. The corresponding ASLan++ code that represents such database is given in Listing 5.6 where: the database is defined as a set of messages `db` (line 1), the table users is also defined as a set of messages `users` (line 2) and the information for the user `bob` is defined with constants (line 3). The table `users` is initialized by including a tuple with the information for the user `bob` (line 3) and the database is initialized by including all the tables composing the database which, in this case, is only one (line 6).

Listing 5.6: ASLan++ code representing the definition of the database for the example in Figure 5.2

```
1 nonpublic db : message set;
2 nonpublic users : message set;
3 nonpublic bob, bobpasswd, bobname, bobsurname, bobphone
      , bobavatar : text;
4 ...
5 users->add((bob, bobpasswd, bobname, bobsurname,
      bobphone, bobavatar));
6 db->add(users);
```

Database operations

Since I do not consider the SQL syntax, I explicitly define the type of queries supported by the formalization of the database, for which I use the following facts:

- `query(Table, Tuple)` (line 1): verifies that `Tuple` is in `Table`,
- `insert(Table, Tuple)` (line 2): inserts `Tuple` inside `Table`,
- `delete(Table, Tuple)` (line 3): deletes `Tuple` from `Table`,
- `update(Table, Oldtuple, Newtuple)` (line 4): deletes `Oldtuple` from `Table` and inserts `Newtuple` in `Table`,
- `query_read(File)` (line 5): interacts with the file-system to read `File`,
- `query_write(File)` (line 6): interacts with the file-system to write `File`.

5.5.2 The behavior of queries

The database entity formalizes a generic database so that this entity can be reused in every scenario where a web application interacts with a database. To formalize the behavior of queries, the database entity defines, within the symbols definition (line 10), variables for: a table (line 10), tuples (line 11) and a file (line 12). I represent the database entity to be always ready to execute new queries (line 15). More specifically, I define the database entity with six `select-on` branches representing the six types of queries supported by my formalization. For each of the six queries, six possible behaviors are formalized: one for the intended behavior (i.e., the Horn clause `attack()` does not hold) and five malicious behaviors (i.e., the Horn clause `attack()` holds). As described in §2.1, a SQLi attack can modify a query to make it behave in a totally different way. I represent this possibility by formalizing in the database entity that whenever any of the previously defined queries and the Horn clause `attack()` hold, a different behavior can be executed. Specifically, attacker is allowed to alter any query and make it behave as one of the following actions (injections):

- SQLi for extracting the content of the database,
- SQLi for creating a tautology,
- SQLi for reading from the file-system,
- SQLi for writing to the file-system,
- SQLi for inserting new content into the database.

When a query needs to be executed (line 18), the database verifies if `Tuple` is present in `Table` (line 20). If that is the case, the database stores in the shared variable `Result` the value of `Tuple`. In the case the Horn clause does hold, five additional behaviors could be performed. I rely on the non-deterministic behavior of the `select-on` statement to let the model checker decide the behavior to perform. The five additional `select-on` define the exploitation of one of the five SQLis supported by my formalization. A SQLi

for creating a tautology forces the database to non-deterministically select a tuple `Sql` contained in `Table` (line 23) and to store it in the shared variable `Result` (line 24). A SQLi for reading the file-system forces the database to non-deterministically select a file contained in the file-system (line 26) and to store it in the shared variable `Result` (line 27). A SQLi for writing to the file-system forces the database to retrieve the file `File` specified in the tuple `Tuple` (line 29) and to add it in the file-system (line 30). The newly created file is then stored in the shared variable `Result` (line 31). A SQLi for inserting a new tuple (line 33) forces the database to include the value `Tuple` into `Table` (line 34). The newly inserted tuple is then stored in the shared variable `Result` (line 35). A SQLi for extracting the entire database (line 37) forces the database to store the constant `db` representing the entire database in the shared variable `Result` (line 38). Finally, the database retracts the fact `query(Table, Tuple)` to notify that the query has been executed (line 41).

When an insert query needs to be executed (line 44), the database can perform a legitimate insert query (line 46) or one of the malicious behaviors previously described (lines 50-62). In case of a legitimate insert query, the database inserts the value `Tuple` into `Table` (line 47) and stores the newly inserted tuple in the shared variable `Result` (line 48). Finally, the database retracts the fact `insert(Table, Tuple)` to notify that the query has been executed (line 66).

When a delete query needs to be executed (line 69), the database can perform a legitimate delete (line 71) or one of the malicious behaviors previously described (lines 75-87). In case of a legitimate delete query, the database deletes `Tuple` from `Table` (line 72) and stores the value of the deleted tuple in the shared variable `Result` (line 73). Finally, the database retracts the fact `delete(Table, Tuple)` to notify that the query has been executed (line 91).

When an update query needs to be executed (line 94), the database can perform a legitimate delete (line 96) or one of the malicious behaviors described previously (lines 101-115). In case of a legitimate update query, the database deletes the old tuple `Oldtuple` from `Table` (line 97) and inserts `Tuple` instead (line 98). The database then stores the value of the newly inserted tuple in the shared variable `Result` (line 98). Finally, the database retracts the fact `update(Table, Oldtuple, Tuple)` to notify that the query has been executed (line 119).

When a query for reading from the file-system needs to be executed (line 122), the database can perform a legitimate query for reading from the file-system (line 124) or one of the malicious behaviors previously described (lines 127-142). In case of a legitimate query for reading from the file-system, the database treats the value `Tuple` as a file and thus verifies whether `file(Tuple)` is contained in the file-system (line 124). If that is the case, the database stores the value `file(Tuple)` in the shared variable `Result` (line 125). Finally, the database retracts the fact `query_read(Tuple)` to notify that the query has been executed (line 146).

When a query for writing to the file-system needs to be executed (line 149), the database can perform a legitimate query for writing to the file-system (line 151) or one of the malicious behaviors previously described (lines 155-169). In case of a legitimate query for writing to the file-system, the database treats the value `Tuple` as a file and thus adds `file(Tuple)` to the file-system (line 152). The database then stores the value `file(Tuple)` in the shared variable `Result` (line 152). Finally, the database retracts the fact `query_write(Tuple)` to notify that the query has been executed (line 173).

5.6 The Web Application

The web application is a node of the network that can send and receive messages. In my formalization, the web application can communicate with the honest client by means of the HTTP protocol and with the file-system and database. The file-system and the database entities do not depend on a specific scenario and thus they can be reused in every model. The web application entity, on the other hand, does depend on the scenario being modeled and thus it is not possible to provide a single entity that can be reused in every model. However, I provide the skeleton of a specification and a series of guidelines on how to represent the web application's behavior for testing the interaction with the client, the file-system and the database.

5.6.1 The HTTP protocol

The Hypertext Transfer protocol (HTTP) is the base communication protocol for interacting with a web application. By its own nature, HTTP is a stateless protocol, which means that each pair request-response is considered as an independent transaction that is not related to any previous request-response. In order to represent this characteristic, I follow the same approach I adopted for the file-system and the database entities, and use `select-on` statements to define that a web application can answer different requests without following a specific sequence of messages.

5.6.2 Client communication

An HTTP request (and response) header comprises different fields that are needed for the message to be processed by the server and the browser. In my formalization, I do not need to represent all the information of a real request (or response) header as they are not relevant for the analysis. I introduce two uninterpreted functions `http_request()` and `http_response()` representing, as the names suggest, an HTTP request and an HTTP response respectively. For an HTTP request I represent:

- a variable for the sender,

- a variable for the receiver,
- a constant for the requested page,
- a concatenation of variables for the parameters, and
- a concatenation of variables for the cookies.

For example, consider the HTTP request in Listing 5.7 where the highlighted texts identify the information represented in the formalization.

Listing 5.7: HTTP request sample

```
GET /index.php?page=menu.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml
    ;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1/index.php
Cookie: PHPSESSID=1234567891011
Connection: close
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
```

The corresponding ASLan++ formalization would be `Client *->* WebApp : http_request(index, Page, Phpsessid)`, where `index` is a constant representing the requested web page, `Page` is a variable representing an HTTP query value and `Phpsessid` is a variable representing the cookie value.

For an HTTP response, I represent:

- a variable for the sender,
- a variable for the receiver,
- a constant for the response page,
- a concatenation of constants, variables and uninterpreted functions for the response body, and
- a concatenation of variables for the cookies.

For example, consider the HTTP response to the previous HTTP requested reported in Listing 5.8.

Listing 5.8: HTTP response sample

```
HTTP/1.1 200 OK
Date: Fri, 13 Oct 2017 08:08:12 GMT
Server: Apache/2.4.25 (Unix) PHP/5.6.30
X-Powered-By: PHP/5.6.30
Pragma: no-cache
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 15348

<!- http code of menu.php ->
```

The corresponding ASLan++ formalization would be `WebApp *->* Client : http_response(index, file(menu), none)` where `index` is a constant representing the name of the responding page (which in this case is the same as the requesting page), `file(menu)` is returned to express that a file was retrieved from the file-system and `none` is a constant representing no cookie is returned to the client.

5.6.3 File-system and database communication.

As already stated in § 5.4, whenever the web application has to read content from the file-system, it uses the predicate `readFile()`, and whenever it has to write to the file-system, it uses the predicate `writeFile()`. When a `readFile()` or a `writeFile()` predicate is used, the web application has to wait for the file-system to retract the appropriate predicate meaning that the operation has been processed and the web application can continue its execution. The result of executing the reading or writing operation is stored in the shared variable `Result`.

Similarly, when the web application has to perform a SQL query on the database, it uses one of the six predicates representing a database query supported by my formalization (see §5.5). The web application waits to be notified that the query has been executed by waiting for the appropriate predicate to be retracted. Once the predicate is retracted, the result of executing the query is stored in the shared variable `Result`. For example, consider a web application performing a query `query(Table, Tuple)` asking the database whether `Tuple` is contained in `Table`. if `Tuple` is contained in `Table`, the database stored the value `Tuple` in `Result` to acknowledge that the value is indeed contained in `Table`. Whenever the web application performs a query the answer provided by the database has to always been forwarded back to the client.

5.6.4 Sessions

As already mentioned, HTTP is a stateless protocol. In order for the user to experience a stateful interaction with a web application (i.e., the web application recognizes when a user is logged in when he changes page), developers make use of HTTP cookies to create sessions (see § 2.3). A *session* allows a web application to store information into a memory area so to have it accessible across multiple web pages. When a request is made to a web page that creates a session, the web page allocates a memory area and assigns to that area a session identifier. The same session identifier is sent back to the client as an HTTP cookie value. When an HTTP cookie is received, a web browser automatically sends it back to the web application when a new request is made. The web application receives the session identifier and uses it to retrieve the information stored within the associated memory area. In order to represent sessions, I introduce a set called `sessions` that contains tuples of the

form `(sessionVal, (key, value))`, where `sessionVal` identifies a session and `(key, val)` maps `key` to `val`.

Consider the example shown in Listing 5.9 where a session value `Phpsessid` is initialized to a fresh value (lines 1). The set `sessions` is then populated with a new tuple associating the variable `Phpsessid` with the tuple `(usersession, User)` (line 2). The value `Phpsessid` is sent back to the client whenever a new session is initialized and whenever a page requires a session, the client has to send a session value that will be checked by the web application and used to retrieve the necessary information.

Listing 5.9: ASLan++ code representing the initialization of a session

```
1 Phpsessid := fresh();
2 sessions->add(Phpsessid, (usersession, User));
```

5.6.5 Remote code execution

The formalization I propose can represent scenarios where the attacker is able to write arbitrary files to the file-system, which might lead to arbitrary remote code execution. As described above, the model of a web application is represented as a sequence of `select-on` statements defining the requests the web application responds to. The possibility of uploading a file that leads to remote code execution can be seen as a way of creating new requests the web application can now respond to. In order to model this possibility, I include into the skeleton model of the web application a series of predefined `select-on` statements representing the behavior of common malicious server side code an attacker might try to upload. I define that these malicious `select-on` can be used by the attacker only if the file exists in the file-system (i.e., `fs->contains (Filepath)` is valid for that file). This will ensure that the attacker finds a way of writing the malicious file before actually using it.

An example representing this behavior is shown in Listing 5.10. A `select -on` statement defines the possibility to handle an HTTP request for the page `evil_file` which can be handled only if `fs->contains(evil_file)` holds (line 2).

Listing 5.10: ASLan++ code representing a remote code execution example

```
1 select{
2  on( Entity*->*Actor: http_request(evil_file, none,
       none).?WebNonce & fs->contains(evil_file)):{
3    % behavior here
4  }}
```

5.7 The honest client

My formalization can address attacks involving the interaction of the DY attacker with an honest client. More specifically, as stated in § 5.3, I propose

a representation of XSS attacks (stored and reflected) and CSRF attacks that require the attacker to interact with an honest client. In the following, I define the representation of an honest client.

The honest client represents the functionalities provided by a web browser. The basic functionality any web browser should be able to perform is to send HTTP requests and handle HTTP responses. Browsers also have an internal memory area where they store private information (e.g., session cookies). Thus, the honest client has to implement a private knowledge representing the new information learned during the interaction with the web application. To represent that, the entity of the honest client defines a private set called `hknows` that represents the knowledge known by the honest client. The content of `hknows` is increased whenever the honest client receives an HTTP response as a result of the interaction with the web application (e.g., cookies values received from the web application after a successful login). Furthermore, browsers provide a JavaScript engine that takes care of executing JavaScript code. As described in § 2.3, JavaScript is often abused by attackers in order to force clients into executing arbitrary client-side code that might compromise the security of the client. I follow the same approach adopted for remote code execution in web application (§ 5.6) and include in the honest client the malicious actions that can be performed as a result of the executing client-side JavaScript. More specifically, the attacker is allowed to steal the knowledge of the honest client by forcing the honest client into sending the set `hknows` to the attacker (i.e., XSS for session hijacking).

The ASLan++ code representing the behavior of the honest client is given in Listing 5.11. Within the symbols definition, the honest client defines a variable used to represent a page (line 3), a variable for representing HTTP parameters (line 4), a variable for representing a cookie (line 5), a variable for representing the body of a response (line 6) and a variable for representing a fresh nonce (line 7). Like the other entities, the honest client is also actively ready to communicate with the web application (line 10). The behavior of the honest client is formalized by two `select-on` branches defining a general request allowing the honest client to interact with the web application (line 12), and the possibility for the attacker to send a message to the honest client and force the execution of an arbitrary request (line 24).

To perform a general request (line 12), the honest client initializes a fresh nonce variable `WebNonce` to ensure a fresh communication (line 13), and it also initializes three variables `Page` (line 14), `Params` (line 15) and `Cookie` (line 16) to a value chosen non-deterministically. Finally, the honest client performs the new request (line 17). When handling the response (line 18), the honest client stores the value `Cookie` into the set `hknows` (line 19) and proceeds by processing the value `Body` that might have been used to carry out an XSS attack. The honest client verifies whether or not the Horn clause `attach()` holds for the value `Body` of the HTTP response (line 21) and, if that is the case, the honest client is forced into sending the set `hknows` to the attacker (line 21).

Finally, the attacker can force the honest client into executing any arbitrary request (i.e., CSRF attack). The honest client receives a message from the attacker with values for the page on which the request should be performed (line 24). The honest client creates a nonce to ensure a fresh communication (line 25) and non-deterministically decides the value for the variable `Cookie` (line 26). An HTTP request is then performed (line 27) and the response is handled the same way previously described (lines 28-31).

Listing 5.11: ASLan++ code representing the behavior of the honest client

```
1  entity Honest(Actor, Webapplication : agent){
2   symbols
3    Page : page;
4    Prams : message;
5    Cookie : cookie;
6    Body : message;
7    WebNonce : nonce;
8
9   body{
10   while(true){
11    select{
12     on(true):{
13      WebNonce := fresh();
14      Page := ?;
15      Params := ?;
16      Cookie := ?;
17      Actor*->*Webapplication:http_request(Page, Params,
            Cookie).WebNonce;
18      Webapplication*->*Actor:http_response(?Page, ?Body
            , ?Cookie).WebNonce;
19      hknows->add(Cookie);
20      if(attack(Body)){
21        Actor *->* i : hknows;
22      }
23     }
24     on( i *->* Actor : ?Page.?Params ):{
25      WebNonce := fresh();
26      select{on(hknows->contains(?Cookie)):{
27       Actor *->* Webapplication : http_request(Page,
            Params,Cookie).WebNonce;
28       Webapplication*->*Actor:http_response(?Page, ?
            Body, ?Cookie).WebNonce;
29       hknows->add(Cookie);
30       if(attack(Body)){
31         Actor *->* i : hknows;
32       }
33       }}
```

```
34        }
35       }
36      }
37     }
38    }
```

5.8 Security properties

The last component of the formalization of a web application, is the enumeration of the security properties (or goals) that should be verified. As discussed in Chapter 5, I am not interested in finding a vulnerability but rather to exploit them. In particular, I am interested in defining security properties related to authentication bypass and confidentiality breach.

Authentication bypass represents the possibility for the attacker to access some part of the web application that should be protected with some sort of authorization mechanisms. Confidentiality represents the possibility for the attacker to obtain information that is "leaked" from the web application. Such "leakage" can happen from either the file-system, the database of the honest client.

I use the LTL "globally" operator `[]`, which defines that a formula has to hold on the entire subsequent temporal path, and the `iknows` predicate in the ASLan++ language, which represents the knowledge of the attacker. Authentication goals can thus being represented by stating that the attacker will never have access to some specific page, and confidentiality goals by stating that the attacker will never increase his knowledge with parts coming from the file-system, the database or the honest client. As an example consider the goal in Listing 5.12 stating that the attacker will never know something of the form `file()` (i.e. will never have access to content stored in the file-system).

Listing 5.12: Confidentiality goal for file inclusion

```
[](!(iknows(file(?))))
```

5.9 Multi Stage case study

I now show how my formalization can be used to create the model of a web application called Multi-Stage[4], which I specifically wrote to show how multiple vulnerabilities can be combined together to generate complex attacks and how WAFEx is able to discover multiple attack traces that violate the same security property. Multi-Stage is depicted in Figure 5.3 as a series of MSC in which there are four entities: honest client, web application, file-system and

[4] Multi-Stage is freely available for testing at [53].

database. Note that the pictures follow standard notational conventions: constants begin with a lower case character (e.g., `username`), variables with an upper case one (e.g., `User`).

I designed Multi-Stage to ensure that it is realistic and representative of software that could indeed be deployed. Multi-Stage provides functionalities that many modern web applications provide to their users; in particular, it provides:

- a registration page that allows users to create an account on the web application (Figure 5.3a);
- an HTTP login page via which users can log in the web application by providing username and password (Figure 5.3b);
- a restricted page that allows logged in users to search for other users (Figure 5.3c);
- a restricted page that allows logged in users to change their own personal information (name, surname, phone number) and to upload an image to use as avatar (Figure 5.3d).

I now describe in more detail the MSC for each functionality provided by Multi-Stage. Figure 5.3a shows the registration process. The user sends a message to the web application containing the constant `register` representing the requested page and the variables `Username` and `Password` representing the credentials for a new use to register to the web application (1). The web application performs an insert request to the table `tableUsers` with the values of `Username` and `Password` (2). Finally, the web application sends a response back to the user containing the constant `index` representing the next page.

In Figure 5.3b is depicted the login process. The honest client sends a message to the web application containing the constant `index` representing the requested page and the variables `Username` and `Password` representing the credentials used to log in the system (1). The web application performs a query request to the database on the table `tableUser` in order to verify that the provided credentials are valid (2). The database returns a response `QueryResponse` to the web application (3). Finally, the web application sends a response back to the honest client containing the constant `index` representing the next page, the response from the database `QueryResponse` and a newly created variable `Phpsessid` representing the value of the session cookie.

In Figure 5.3c is depicted the MSC for search users in the web application. The honest client sends a message to the web application containing the constant `search` representing the requested page and the variables `Userid` and `Phpsessid` representing, respectively, the honest client to search and the value of the session cookie (1). The web application checks if `Phpsessid` is a valid session value (2) and, if that is the case, it performs a query to the database for search the user represented by `Userid` (3). The database answers to the web application by sending the result of executing the query (4) and the web application sends a response back to the honest client containing the constant `search` identifying the current page, the value `Userid` representing

the honest client requested and the value `QueryResponse` representing the response of the search query.

In Figure 5.3d is depicted the possibility to edit perform information. The honest client sends a message to the web application containing the constant `profile` representing the requested page, the variables `Name, Surname, Phone, File_avatar` representing, respectively, name, surname, phone number and an image file to use as avatar and the variable `Phpsessid` representing the value of the session cookie (1). The web application checks if `Pphpsessid` is a valid session value (2) and, if that is the case, it performs a query request to the database for changing the values of name, surname, phone number and avatar (3). The database answers to the web application with the variable `QueryResponse` (4). The writes to the file-system the newly uploaded file for the avatar (5). Finally, the web application sends a response back to the honest client containing the page `profile` and the response from the database `QueryResponse`.

5.9.1 The specification

The ASLan++ code representing the web application entity for the Multi-Stage case study is given in Listing 5.13. The web application entity can communicate with the honest client, the database and the file-system. The web application defines symbols required for the formalization of Multi-Stage in the symbols definition (lines 3-8) and actively listens for incoming connections (line 11). It can answer to four different requests for: registering new users (line 14), logging in users (line 22), searching users in the database (line 36) and updating personal information (line 44). Additionally, the model includes two malicious server-side requests that could be exploited by the attacker (as described in § 5.6). The first malicious request allows the attacker to access the entire database (line 57), whereas the second allows the attacker to access the entire file-system (line 60). Clients can register new users to the web application by means of the registration request, where they provide values for `User` and `Password` (line 14). The web application then generates a tuple `Tuple` filling the remaining fields with random values (line 15) and, by means of the `insert()` predicate, performs an insert query on the database (line 16). The web application then waits until the insert query has been executed (line 17) and answers to the client by sending `Result`, representing the result of executing the query (line 18).

The login request allows users of the web application to log in by providing proper credentials `User` and `Password` (line 22). The web application creates a tuple to query the database (line 23) and, by means of the `query()` predicate, performs a query on the database (line 24). The web application then waits for the query to be executed (line 25) and verifies that a proper user has been retrieved as result (line 26). If that is the case, the execution proceeds, the web application creates a new session and sends the session to the communicating entity along with the constant `dashboard` representing the next page

and `Result` representing the result of the query (lines 27-29). If the result of the query is not a proper user in the database, the web application does not create a session and simply answers by sending `Result`, representing the result of executing the query (line 31).

A search page allows users to search for other registered users. The search functionality is accessible only by logged in users. web application. The web application receives the variable `Search` that identifies the user to search for and the variable `Phpsessid` that identifies a cookie value (line 36). The web application verifies that the variable `Phpsessid` identifies a valid cookie value and grants access to the page (line 36). If the user is granted permission to access the search page, the web application creates a tuple to query the database (line 37) and queries the database (line 38). The web application waits for the database to process the query (line 39) and answers client by providing the result of executing the query `Result` concatenated with the variable containing the requested value `Search` (line 40). The last functionality provided by the Multi-Stage web application is the possibility for a logged in user to update their personal information. The web application receives a request for the page `profile` along with the parameters `Name`, `Surname`, `Phone`, `Avatar` and the cookie value `Phpsessid` (line 44). The web application then verifies that `Phpsessid` is a valid cookie value and retrieves from the session identified by `Phpsessid` the name of the logged in user that will be used in the update query (line 44). The update does not allow one to change the password, thus this value should be preserved. In order to do so, the web application first creates a query for retrieving the tuple already stored in the database (lines 45-46) and then creates a new tuple that will only update the values for name, surname, phone number and avatar (line 47). The variable `Var_1`, which is used to store the value of the password in the database, is reused when creating the new tuple since the password cannot be changed. The web application then performs the update query (line 48) and waits for the query to be performed (line 49). Once the update has been performed, it is time to upload the value of the avatar field representing the image of the user. The web application performs a write operation on the file-system (line 50) and waits of the operation to complete (line 51). Once the write is completed, the web application answers to the update request by sending the variable `Result`, representing the result of the update query (line 52).

Finally, the model of the Multi-Stage web application handles two additional requests representing the possibility for the attacker to access the entire file-system (lines 60-61) or the entire database (lines 57-58), provided that he managed to upload a malicious file represented by the constant `evil_file`.

Listing 5.13: ASlan++ code for the Multi-Stage case study

```
1  entity Webapplication(Actor, Honest : agent) {
2   symbols
3    Entity : agent;
```

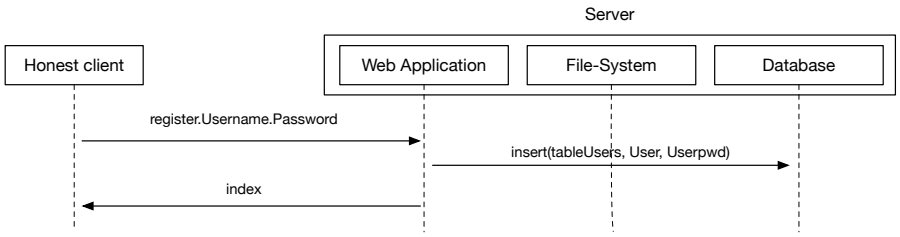
```
4    Var_1, Search, User, Usertm, Password, Name, Surname,
         Phone : param;
5    Avatar: fnode;
6    Tuple, Oldtuple : message;
7    Phpsessid : cookie;
8    WebNonce : nonce;
9
10   body{
11    while(true){
12     select{
13      % registration
14      on(?Entity*->*Actor:http_request(register, ?User.?
            Password, nonec).?WebNonce):{
15       Tuple := User.Password.?.?.?.?;
16       insert(users, Tuple);
17       select{ on(!insert(users, Tuple)):{
18        Actor*->*Entity:http_response(index, Result,
             nonec).WebNonce;
19       }}
20      }
21      % login
22      on(?Entity*->*Actor:http_request(index, ?User.?
            Password, nonec).?WebNonce):{
23       Tuple := User, Password,?,?,?,?;
24       query(users, Tuple);
25       select{ on(!query(users, Tuple)):{
26        if(Result = (?User,?Password,?Name,?Surname,?
             Phone,?Avatar)){
27         Phpsessid := fresh();
28         sessions->add((Phpsessid, usersession, User));
29         Actor*->*Entity:http_response(dashboard, Result,
              Phpsessid).WebNonce;
30        }else{
31         Actor*->*Entity:http_response(index, Result,
             nonec).WebNonce;
32        }
33       }}
34      }
35      % search
36      on(?Entity*->*Actor:http_request(search, ?Search, ?
            Phpsessid).?WebNonce & sessions->contains((?
            Phpsessid, usersession, ?Usertm))):{
37      Tuple := Search.?.?.?.?.?;
38      query(users, Tuple);
39       select{ on(!query(users, Tuple)):{
40           Actor*->*Entity:http_response(search, Search.
                Result, nonec).WebNonce;
41       }}
42       }
```

```
43      % profile update
44       on(?Entity*->*Actor:http_request(profile, ?Name.?
             Surname.?Phone.?Avatar, ?Phpsessid).?WebNonce &
              sessions->contains((?Phpsessid, usersession, ?
             User))):{
45       Var_1 := ?;
46       Oldtuple := (User,Var_1,?,?,?,?);
47       Tuple := User.Var_1.Name.Surname.Phone.Avatar;
48       update(users, Tuple, Oldtuple);
49       select{ on(!update(users, Tuple, User.Var_1.Name.
             Surname.Phone.Avatar)):{
50         writeFile(Avatar);
51          select{ on(!writeFile(Avatar)):{
52            Actor*->*Entity:http_response(dashboard,
                  Result, nonec).WebNonce;
53         }}
54       }}
55       }
56
57      on(?Entity *->* Actor : http_request(evil_file,
           none, nonec) & fs->contains(file(evil_file)) ):{
58       Actor -> i : db;
59      }
60      on(?Entity *->* Actor : http_request(evil_file,
           none, nonec).?WebNonce & fs->contains(file(
           evil_file)) ):{
61       Actor -> i : fs;
62      }
63     }
64    }
65   }
66  }
```

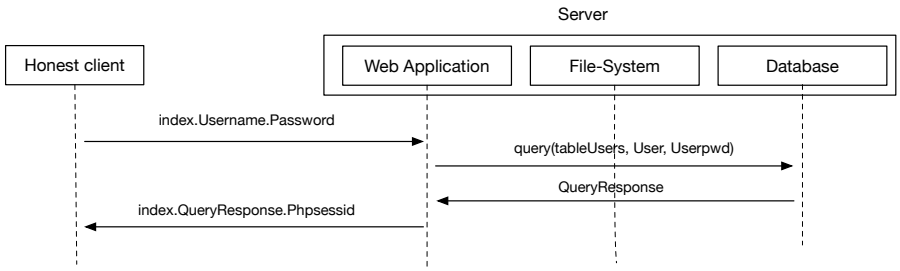
5.10 Conclusions

In this chapter, I have described my formalization for modeling vulnerable web applications. I have shown how the canonical DY attacker model can be used to exploit vulnerabilities of web applications. Moreover, I have formalized four entities: the file-system, the database, the web application and the honest client; representing the vulnerable parties of a web application. I have described how to express security properties in terms of authentication bypass and confidentiality breach along with the description of Multi-Stage, a case study that shows how my formalization ca be used to model a real web application.

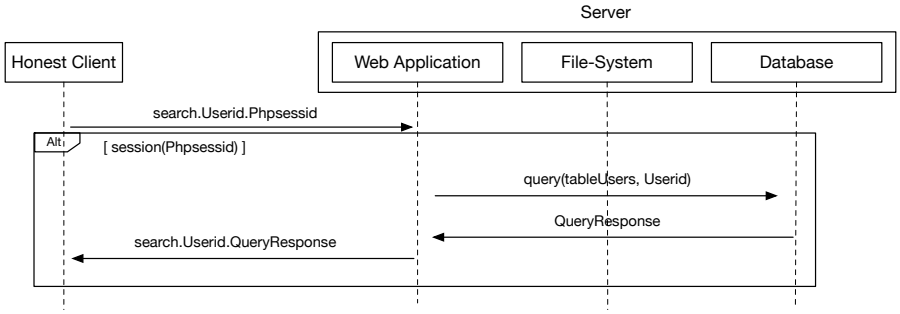
Fig. 5.3: The MSCs of the Multi-Stage case study



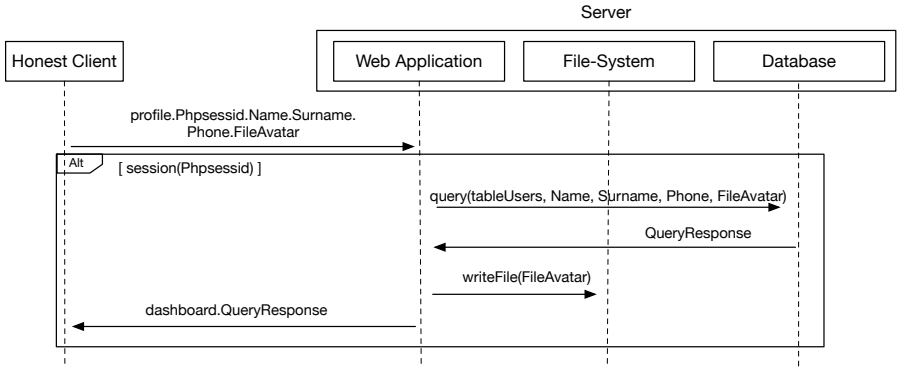
(a) MSC representing the registration procedure



(b) MSC representing the login procedure



(c) MSC representing the possibility to search users



(d) MSC representing the possibility to edit personal information

6

WAFEx

I have developed, using the Python3 language, a prototype tool called *Web Application Formal Exploiter (WAFEx [86])* which implements my approach. The purpose of WAFEx is to help the security analyst during the exploitation phase of a penetration test described in §3.2.1.2. Specifically, WAFEx can help in identifying how to exploit vulnerabilities of web applications by generating multi-stage attacks. As depicted in Figure 1.1, WAFEx implements a model-based testing approach that consists of two phases: a *model creation* phase and a *concretization* phase. In the model creation phase the security analyst creates a model of a web application while in the concretization phase the model of a web application is analyzed and tested on the real web application. I applied WAFEx to a number of case studies that are discussed at length in § 6.3. Before describing the case studies, I describe the two phases in more detail.

6.1 Model creator

The model creation phase implemented in WAFEx consists of a plug-in for the Burp Proxy [68] (Burp, for simplicity) that helps the security analyst in the creation of a specification in ASLan++. The security analyst records an HTTP trace by using Burp (① in Figure 1.1) and with the *Model Creator Plugin [87]* generates an ASLan++ skeleton of the web application along with a concretization file (② in Figure 1.1). I have chosen ASLan++ so as to be able to apply the model checkers of the AVANTSSAR Platform [6] (in particular, CL-AtSe [80]), but my approach is general and could be quite straightforwardly used with other specification languages and/or other reasoners implementing the DY attacker model. The skeleton of the ASLan++ model is created following the formalization I presented in Chapter 5, while the concretization file is meant to tie together abstract requests in ASLan++ with details needed to perform the real requests on the web application. In

order to do so, I performed some minor changes to the ASLan++ formalization described in Chapter 5. More specifically, I performed the following two changes:

- Every HTTP request is associated with a progressive constant `tag#` that is used to identify the details of the request in the concretization file.
- Instead of having one constant `malicious` that represents every possible malicious input, I use different constants to better identify the payload that should be used to attack the web application. In particular, I use the following constants instead:
  - `sqli`: SQLi payload for extracting the entire database,
  - `sqli_bypass`: SQLi payload for creating a tautology,
  - `sqli_read`: SQLi payload for reading from the file-system,
  - `sqli_write`: SQLi payload for writing to the file-system,
  - `fsi`: file-inclusion payload for reading from the file-system,
  - `xss`: XSS payload for stealing the user's session.

The concretization file is represented in the JSON data format and its structure is shown in Listing 6.1. For every request in the ASLan++ model, an identification tag is created in the concretization file (line 2). The details of a request are:

- the HTTP method used to perform the request (line 3),
- the real URL of the request (line 4),
- the parameters used in `GET` request, that are saved in a JSON object (line 5) pairing the abstract parameter (i.e., the one used in the model) with a JSON array of two elements representing the real key and associated value (line 6),
- the parameters used in `POST` request, that are represented just like the get parameters (line 8),
- the cookies that are also represented as a JSON object (line 9) mapping an abstract cookie with the corresponding pair consisting of key and value (line 10).

Listing 6.1: JSON structure of the concretization file

```
{
 "tag#": {
  "method": "[GET|POST]",
  "url": "http[s]://[url]",
  "get_params":{
   "[abstract_param]":["real_param","real_value"]
  },
  "post_params":{}
  "cookies": {
   "[abstract_cookie]": ["real_cookie_key","
      real_cookie_val"]
```

```
    },
  }
}
```

As shown in Figure 6.1, the Model Creator plug-in extends the usual options provided by Burp by including a new button `Send to WAFEx`. The security analyst can then select the requests\responses he wants to use for creating the model and then, by means of the `Send to WAFEx` button, they can be sent to the WAFEx model creator interface (Figure 6.2).

The Model Creator plug-in graphical interface can be used to edit the skeleton of an ASLan++ model and the concretization file. On the left side of the interface there are two panels ((1) and (2) in Figure 6.2) that shows the requests to be converted to an ASLan++ model. The first panel ((1) in Figure 6.2) shows a table where rows represent a single pair request\response. By selecting one raw, the second panel ((2) in Figure 6.2) shows the details of the selection. Specifically it can show the details of both the request or the response. This two areas can be helpful for the analyst in order to always have an overview of the real requests and responses he has to model. WAFEx model creator allows the security analyst to specify an SQL file that represent the database used by the web application ((3) in Figure 6.2). The SQL file is then used to automatically create the database structure of the ASLAn++ model by following the formalization I propose in § 5.5. Once the requests\responses and the SQL file has been selected, the security analyst can press the `Generate!` button in order to generate the model. Once the model is generated, it is displayed in the main are consisting of a syntax highlighted text area that allows the modeler to complete the newly created model ((4) in Figure 6.2). WAFEx model creator also creates the concretization file and provides the possibility to edit it by selecting the corresponding tab ((5) in Figure 6.2). Once the model is ready, it can be easily saved thanks to the `Save` button. Once the ASLan++ model and the concretization file are ready, the concretization phase can start.

6.2 Concretization

Once the ASLan++ model and the corresponding concretization file are ready, the analysis can start. The security analyst interacts with WAFEx (③ in Figure 1.1) to start the testing of a web application. WAFEx makes use of the model checker CL-AtSe [80] to analyze an ASLan++ model (④ in Figure 1.1) and, in case an attack is found, CL-AtSe generates an AAT represented as a MSC (④ in Figure 1.1). An AAT shows the requests that, when executed, violates the security property defined on the model of the web application. However, since the model of the web application represents an abstraction of the real web application, it is required to test the AAT on the real web application to ensure that the attack found can actually be carried out. WAFEx

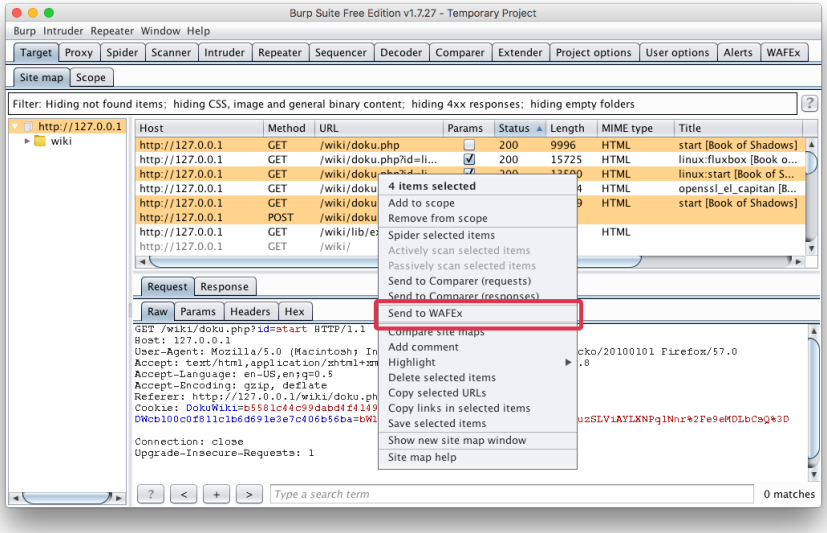


Fig. 6.1: WAFEx model creator: button for selecting requests and responses to process

does so automatically by reading the AATs, along with the concretization file (⑥ in Figure 1.1), and performs the attack on the real web application (⑦ in Figure 1.1). WAFEx is then capable of understanding what kind of request should be performed at a given step of an AAT thanks to the additional details in the ASLan++ model that I described in § 6.1. In case a vulnerability has to be exploited, WAFEx uses state-of-the-art tools such as Wfuzz [31] and sqlmap [74] for the generation of a malicious payload, otherwise it uses the information in the concretization file to perform the request.

It is worth noting that CL-AtSe does not allow for the generation of multiple AATs (nor do the other back-ends of the AVANTSSAR Platform). Thus, whenever a trace was found in the analysis of the case studies, I disabled the branch corresponding to the attack in the `select-on` for the entity that was used in that trace and run WAFEx again to generate another trace different from the previous one (if such a trace exists). This process does, of course, miss some traces since disabling a branch prevents any other trace to use that branch in a different step of the attack trace.[1] However, it shows that multiple traces can actually be generated.

The parameters and usage details of WAFEx are shown in Listing 6.2.

[1] I plan to extend CL-AtSe or replace it with a tool capable of generating multiple attack traces.

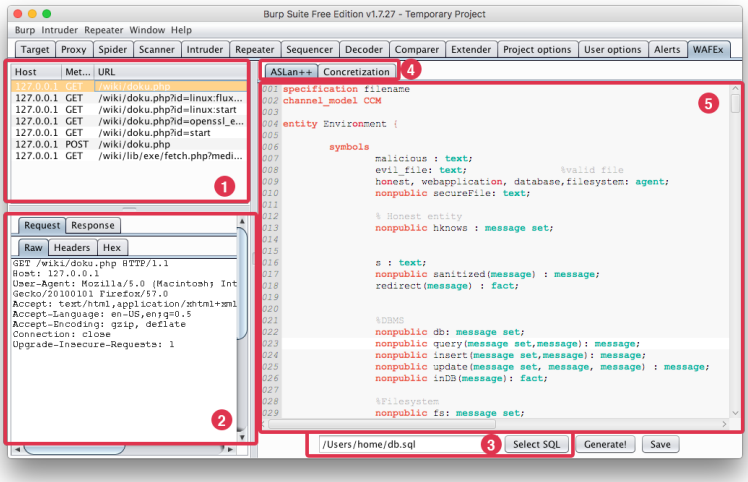


Fig. 6.2: WAFEX model creator: main interface area

Listing 6.2: WAFEx usage parameters

```
usage: wafex.py [-h] [--c concre_file] [--debug]
                [--mc-only] [--interactive]
                [--verbose] [--translator]
                [--proxy ip:port]
                [--mc-options MC_OPTIONS]
                [--mc-timeout T]
                model

positional arguments:
  model                 An ASLAn++ model

optional arguments:
  -h, --help            show this help message and exit
  --c concre_file       The concretization file, needed
                        for executing Abstract Attack
                            Trace
  --debug               Print debug messages
  --mc-only             Run the model checker only and
      exit
  --interactive         Ask input of every parameter
  --verbose             Increase the output verbosity

Translator:
  --translator          Specify a jar translator to use.
                        Allowed values are 1.4.1, 1.4.9,
```

```
                        1.3. Default (1.4.1)

  HTTP(S) options:
    --proxy ip:port     Use an HTTP proxy when executing
                        requests

  Cl-Atse options:
    --mc-options MC_OPTIONS
                        String representing the options
                        to pass to Cl-Atse.
                        For more information on the
                        available options check
                        Cl-Atse manual
    --mc-timeout T      If Cl-Atse runs more than T
                        seconds, abort (default: 600)
```

6.3 Experimental results

In this section, I show how my formalization can be used effectively for representing and testing attacks involving the exploitation of multiple web application vulnerabilities. During the development of WAFEx, I first applied it to some well-known vulnerable web applications to check whether WAFEx was able to identify the vulnerabilities I formalized. In particular, I applied WAFEx to: DVWA [30], WebGoat [63] and Gruyere [44], which provide state-of-the-art environments for pentesters to improve their skills and tools[2]. WAFEx has been able to correctly identify all the vulnerabilities described in Chapter 2, which allowed us to refine the concretization phase of WAFEx to deal with the actual exploitation. After this initial testing, I took a step further and considered two concrete and complex case studies in order to show that my approach can indeed be applied to real web applications but more importantly that WAFEx can be used to perform an analysis that no other tool is currently capable of. Specifically, I applied WAFEx first to the Multi-Stage case study that I presented in § 5.9 and then to Cittadiverona (`http://www.cittadiverona.it`), a web application that collects, organizes and shows local events that take place in the city of Verona, Italy. Cittadiverona is a customized version of the Joomla! CMS that was developed over a number of years by Virtuopolitan S.r.l. and previous owners. Cittadiverona provides a concrete and generic example of the security issues of a real-world web application.

6.3.1 Case study: the Multi-Stage web application

I already described the Multi-Stage web application in § 5.9 as a means to illustrate how to create a model of a web application. In this section, I show

[2] See § A.1 for details on the ASLan++ model for these case studies.

the result of applying WAFEx to Multi-Stage. I defined the security property given in Listing 6.3, that wants to ensure that the attacker is not able to have access to the entire database, and CL-AtSe then generated four different AATs that violate this property.

Listing 6.3: Confidentiality goal for the Multi-Stage case study

```
[](!(iknows(db)));
```

6.3.1.1 The abstract attack traces

Abstract Attack Trace #1

This first AAT (Figure 6.3) shows a simple attack where the attacker might be able to exploit a SQLi in the login phase to directly access the entire database. The attacker sends to the web application a request for the page `index` by sending the constant `malicious` and `Password(84)` as login credentials, and the constant `nonec` meaning no cookie is sent (①). The web application, when performing the query for checking the credentials (②), is forced into executing one of the malicious actions I formalized in § 5.5. In this case, the database answers to the login query with the content of the database `{{bob.bobpasswd.bobname.bobsurname.bobphone.bobavatar}}` (③) meaning that the attacker used a SQLi attack to dump the entire database. The result of executing the query is then sent back to the attacker as a response to the login phase (④).

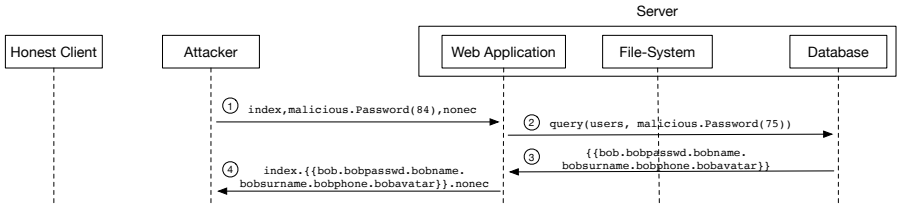


Fig. 6.3: AAT #1 for accessing the database in the Multi-Stage case study

Abstract Attack Trace #2

I disabled the branch that allows the attacker to force the dump of the entire database and ran the model checker again to generate a different AAT (Figure 6.4). The attacker sends to the web application a request for the page `index` by sending the constant `malicious` and the constant `evil_file` as login credentials (①). The web application performs a query to the database

(②) and, because of the constant `malicious`, the database is forced into executing one of the malicious actions I formalized in § 5.5. In this case, the constant `malicious` triggers the possibility of directly writing to the file-system. The file being written is the file represented by the constant `evil_file` (③). Since the write operation does not return any value, the web application simply answers to the attacker with `dummy_message` (④). The attacker can now use the `evil_file` to have direct access to the database (⑤ and ⑥).

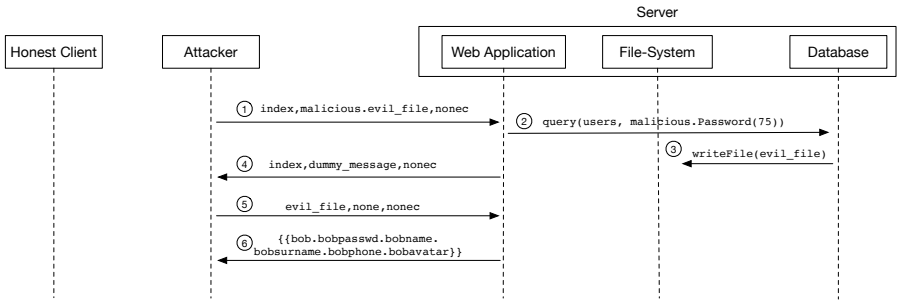


Fig. 6.4: AAT #2 for accessing the database in the Multi-Stage case study

Abstract Attack Trace #3

I disabled the branch that allows the attacker to force the database into writing to the file-system and ran the model checker again to generate a different AAT (Figure 6.5). The attacker sends to the web application a request for page `index` by sending the constant `malicious` and `Password(75)` as login credentials (①). Once again the database is forced into executing one of the malicious actions I formalized in § 5.5, which in this case would be to bypass the login process and have access without knowing correct credentials (② and ③). The attacker thus gains access as the user `bob` (④), who is a legitimate user in the web application. The attacker can now take advantage of the page `profile` that gives the possibility of updating the personal information of the user currently logged in. The attacker exploits this functionality to upload a remote shell as an avatar image (⑤). The request to page `profile` causes the execution of a query that updates the details of user `bob` (⑥) and returns a tuple to the web application with the new details (⑦). The request to page `profile` also causes the upload to the file-system of the file `evil_file` (⑧). The web application then answers to the attacker with the updated details of user `bob` (⑨). Finally, like in the previous AAT, the attacker can now take advantage of `evil_file` (⑩) to have direct access to the database (⑪).

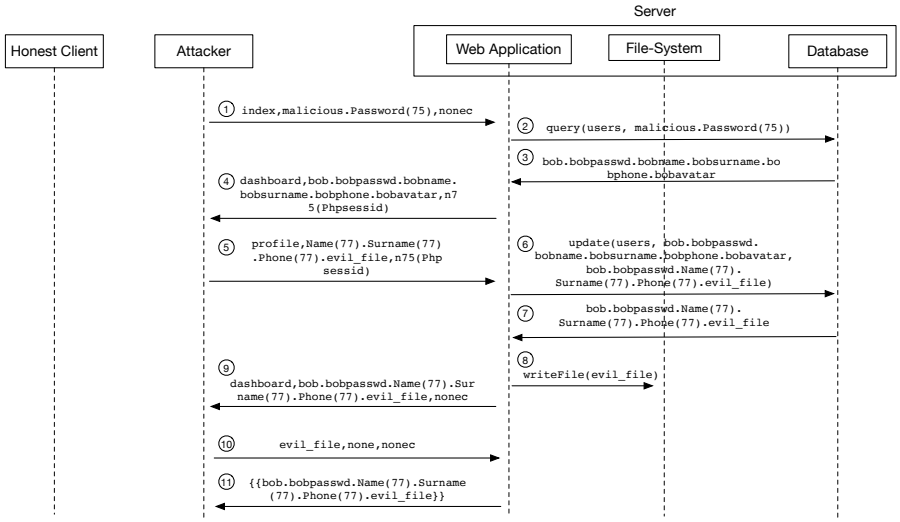


Fig. 6.5: AAT #3 for accessing the database in the Multi-Stage case study

Abstract Attack Trace #4

I generated a fourth, more complex AAT showing how the attacker can have access to the database even when no SQLi attack is possible. I removed the possibility for the attacker to perform any kind of SQLi on the database and generated the AAT in Figure 6.6. In this AAT the interaction starts with the honest client sending a request for page `index` providing credentials for the user bob (`bob` and `bobpasswd` ①). The web application performs a query on the database (②) and answers with bob's details (③). The honest client is then granted access to the web application and the session value `n83(Phpsessid)` is sent back to him along with the result of executing the login query (④). The attacker now performs a CSRF attack on the honest client by sending to him a malicious request for page `search` along with the constant `malicious` (⑤). The CSRF attack forces the honest client into performing a request to the page `search` using the constant `malicious` as parameter (⑥). The `search` page is used to search a user in the database, thus the web application performs a query to the database using the constant `malicious` (⑦). The execution of such a query does not produce any tuple, and therefore the constant `none` is returned by the database (⑧). When page `search` answers to a request, it reflects the value of the parameter searched along with the result of the query. The web application then answers to the honest client by sending back the constant `malicious` along with the result of executing the query, which in this case is `none` (⑨). This is where the honest client receives the constant `malicious` and the CSRF attack actually becomes a reflected XSS attack. Recall that based on my formalization (§ 5.7), whenever

the honest client receives the constant `malicious`, he is forced into sending his knowledge to the attacker. The honest client thus sends his knowledge, represented by `{n83(Phpsessid),nonec}`, to the attacker (⑩). The attacker can now use the session values `n83(Phpsessid)` to impersonate user `bob` and, just like in the previous AAT, the attacker can abuse the possibility of uploading an avatar in order to upload a malicious code represented by the constant `evil_file` (⑪—⑮). Finally, the attacker can take advantage of `evil_file` and access the entire database (⑯ and ⑰).

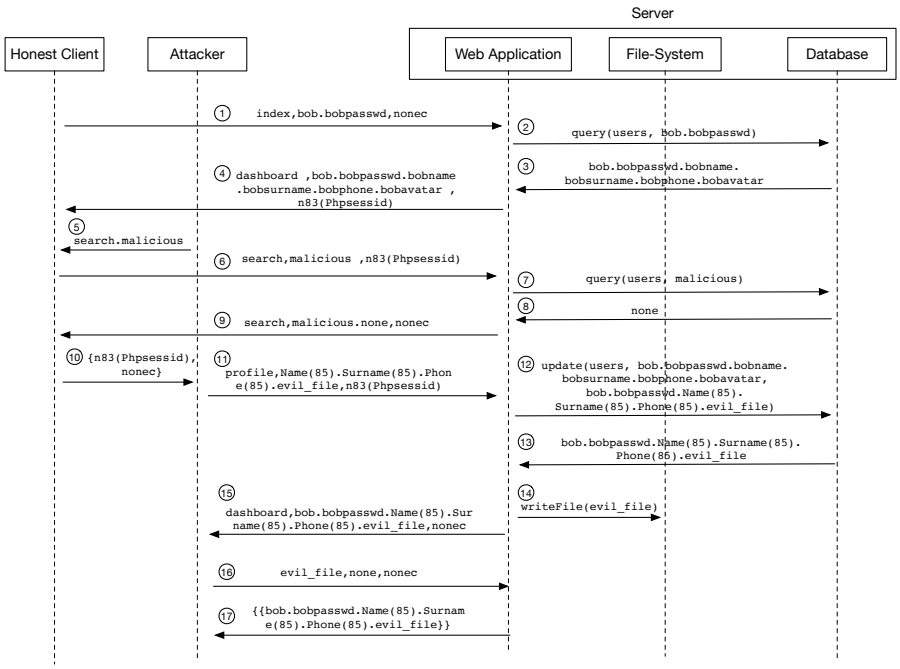


Fig. 6.6: AAT #4 for accessing the database in the Multi-Stage case study

6.3.1.2 Concretization

For each AAT generated for the Multi-Stage case study, WAFEx performed the concretization phase trying to attack the real web application. I run WAFEx on a Mac Book laptop (Intel i5-4288U with 8G RAM and Python3.5). The execution time of the model-checking phase to generate an AAT ranges from 30ms to 50ms. Of the four AATs, WAFEx was able to concretize AAT #1, #3 and #4. It was not possible to concretize AAT #2 since the user that was running the database did not have writing privileges on the file-system and thus it was not possible to exploit a SQLi for writing on the file-system.

6.3.2 Case study: Cittadiverona

Cittadiverona is a web application developed and maintained by Virtuopolitan S.r.l., a communication agency focused on the development and managing of touristic web applications. Cittadiverona has been active since 2005 and is now the main point of reference for locals and foreigners who are looking for information about the many events offered by different organizations in the city of Verona (Italy) and its surroundings. It currently has 969.000 registered users and around 15 new events are posted every week.

Cittadiverona is a web application based on the Joomla! CMS that over the years has gone through many changes that were required to maintain the web application functioning and meet new requirements. Some of the changes made to Cittadiverona allowed it to be highly ranked on Google. Cittadiverona provides features similar to the ones that I described in the case study § 5.9 plus other features specific to Cittadiverona's needs and goals. Specifically, Cittadiverona provides:

- a public registration page that allows users to create an account on the web application;
- a public HTTP login page that allows users to log in the web application by providing username and password;
- a public page that users can use to query the database looking for events — users can search events based on different filters (e.g., description, time period, location);
- a public page for displaying details of a specific event;
- a restricted page where only logged in users can submit new events; once the user sends a new event, the administrators can view and edit it.

Furthermore, Cittadiverona provides administrative functionalities that are accessible only to specific users identified as administrators:

- a page where all events can be viewed and edited;
- a file management area that can be used to upload new files and read from the file-system;
- a backup page that can be used to save the entire content of the database and the file-system;
- a user-management area where users can be created and edited.

The first phase of the analysis was to create a model of the Cittadiverona case study. I then used the Model Creator plug-in to create the skeleton of the web application (see § 6.1) and I manually refined the skeleton to reflect the functionalities offered by Cittadiverona. Listing 6.4 shows the resulting ASLan++ specification.

6.3.2.1 The specification

Administrators are allowed to log in using a dedicated web page `adminlogin` by providing proper credentials identified by `Var_0` and `Var_1` (line 14); note that

variables' names are generated automatically by the Model Creator Plugin. Once credentials are submitted, the web application creates a tuple (line 15) and performs a query on the table `jos_users` (line 16). The web application now waits until the query is executed (line 17) and, if the result of executing the query (stored in `Result`) is of the right format (line 18), then it verifies that the corresponding `User` returned by the query is indeed a valid administrator (line 19). The predicate `isAdmin(agent)` is used to identify which agent in the model has administrative privileges. If the value of `User` identifies a valid administrator, the web application creates a session and associates the `User` to it (lines 20 and 21). Finally, the web application answers to the login request by sending back the result of executing the query and the newly created session (line 22). If the result of executing the query does not produce a valid user (i.e., the if statement in line 18 evaluates to false), the web application does not create a new session and responds by sending the result of executing the query (line 25).

Administrators can perform a series of actions. In particular, they can show events saved in the database (line 30). The web application receives a request for page `adminindex` with a variable `Var_0` identifying the id of the event to show and a variable `Sessid` identifying a session value. The web application verifies that the value of `Sessid` is indeed a valid session value and that the user associated to that session is an administrator (line 30). If that is the case, the web application grants access to the page, creates a tuple (line 31) and queries the database (line 32). The web application waits for the query to be executed (line 33) and answers with the result of executing the query (line 34).

Administrators can also edit events. The web application receives a request for page `adminindex` with variables `Var_0`, `Var_1`, `Image`, `Var_3`, `Var_4`, `Var_5` and `Var_6` representing, respectively, the title, description, picture, category, place, period and identifier of the event to edit. Furthermore, the request contains the variable `Sessid` that the web application uses to verify that the request comes from a logged in user that has administrative privileges (line 53). The web application first initializes the variable `Oldtuple` representing the old tuple that will be updated (line 54) and then initializes the variable `Tuple` representing the new tuple (line 55). The web application performs the update query (line 56) and waits for the database to complete the operation (line 57). Once the operation is completed, the web application uploads the value of `Image` used to represent a picture associated with the event being edited. To do so, the web application performs a write operation on the file-system (line 58) and waits for the operation to be completed (line 59). Once the write operation is completed, the web application answers with the result of executing the update query (line 60).

Administrators can also upload new files to the file-system. The web application receives a request for the page `adminindex` along with a variable `File` representing the file to upload and a variable `Sessid` representing a session value (line 65). The web application verifies that the value for `Sessid` is a

valid session value and that the associated agent has administrative privileges (line 65). If that is the case, the web application performs a write operation (line 66) and waits for the file-system to perform the operation (line 67). Once the operation is completed, the web application does not need to send an answers since the write operation on the file-system does not return a result (as formalized in § 5.4).

Administrators can also read content from the file-system. The web application receives a request for the page `adminindex` along with a variable `File` representing the file to read and a variable `Sessid` representing a session value (line 72). The web application verifies that the value for `Sessid` is a valid session value and that the associated agent has administrative privileges (line 72). If that is the case, the web application performs a reading operation (line 73) and waits for the file-system to complete the operation (line 74). Once the operation is completed, the web application answers by sending the value of the variable `Result` containing the result of the reading request performed on the file-system (line 75).

Administrators can create new users (line 38) and edit user details (line 45). Creating a new user is achieved by sending a request to page `adminindex` along with variables `Var_0`, `Var_1`, `Var_2` and `Var_3` representing, respectively, the username, password, name and email address of the new user. The request contains also the variable `Sessid` representing a session value (line 38). The web application verifies that the value for `Sessid` is a valid session value and that the associated agent has administrative privileges (line 38). If that is the case, the web application creates a tuple representing the new user to add to the database (line 39) and performs an insert query to add `Tuple` to the table `jos_users` (line 40). The web application then waits for the insert query to complete (line 41) and then answers by sending the variable `Result` representing the result of executing the insert query (line 42). For editing users, the web application receives a request for page `adminindex` along with variables `Var_0`, `Var_1`, `Var_2`, `Var_3` and `Var_4` representing, respectively, the username of the user to edit, the new username, new password, new name and new email address of the user. The request contains also the variable `Sessid` representing a session value (line 45). The web application verifies that the value for `Sessid` is a valid session value and that the associated agent has administrative privileges (line 45). If that is the case, the web application initializes the value `Oldtuple` that represents a database tuple for the user to be edited, which is identified by the variable `Var_0` (the additional `?` are used to define the arity of the tuple) (line 46). The web application then initializes `Tuple` that is going to replace `Oldtuple` (line 47) and performs the update query (line 48). The web application waits for the update query to complete (line 49) and then answers by sending the variable `Result` representing the result of executing the update query (line 50).

Finally, administrators can retrieve the entire database and file-system as part of a backup functionality. The web application receives a message for the page `admnistrator_backup_php` and the variable `Sessid` representing a session

value (line 79). The web application verifies that the value for `Sessid` is a valid session value and that the associated agent has administrative privileges (line 79). If that is the case, the web application answers by sending the sets `fs` and `db` representing, respectively, the file-system and the database (line 80).

Normal users can register to the web application by sending a request to the page `register` along with variables `Var_0`, `Var_1`, `Var_2` and `Var_3` representing, respectively, username, password, name and email of the new user (line 98). The web application creates a tuple with the details of the new user (line 99) and performs an insert query to the database (line 100). Once the insert query is completed (line 101), the web application answers by sending the result of executing the query (line 102).

Registered users can log in by sending a request to the page `login` with variables `Var_0` and `Var_1` representing, respectively, username and password (line 84). The web application creates a tuple to verify the credentials (line 85) and queries the database (line 86). The web application then waits for the operation to complete (line 87). If the result of the query contains a valid user (line 88), the web application creates a session (line 89) and associates to it the agent retrieved from the database (line 90). Finally, the web application answers by sending the result of executing the query along with the newly created session (line 91). If the result of executing the query does not contain a valid username, the web application answers with the result of executing the query but without initializing a valid session (line 91).

Users that are not logged in the web application can query the database to search for events. The web application receives a request for the page `index` with variables `Var_0`, `Var_1`, `Var_2` and `Var_3` (line 118) representing, respectively, title, description, category, location and time period of the event. The web application then creates a tuple to query the database (line 119) and performs a query operation (line 120). Once the query operation is completed (line 121), the web application answers by sending the result of executing the query (line 122).

Logged in users can insert new events in the database by sending a request to the page `addevent` with variables `Var_0`, `Var_1`, `Var_2`, `Var_3`, `Var_4` and `Image` representing, respectively, title, description, category, time period, location and image for the event to insert. Within the request, the variable `Sessid` represents a session value. The web application verifies that the `Sessid` is indeed a valid session value and grants access to the page. The web application then generates a fresh value for the variable `Var_6` that will be used as identifier for the new event (line 107) and creates a tuple with the details of the new event (line 108). The web application performs an insert operation on the database (line 109) and waits for the database to complete the operation (line 110). Once the insert query is completed, the web application uploads the value of the variable `Image` to the file-system by performing a write operation (line 111). The web application waits for the file-system to complete the write operation (line 112) and finally answers by sending the result of executing the insert query (line 113).

Finally, the specification of the Cittadiverona web application handles two additional requests for allowing remote code execution as formalized in Chapter 5. The first requests allows the attacker to access the entire file-system (lines 127-128) and the second allows the attacked to access the entire database (lines 130-131) provided that he managed to upload a malicious file identified by `evil_file`.

Listing 6.4: ASLan++ specification for the Cittadiverona case study

```
1  entity Webapplication(Actor, Honest : agent) {
2   symbols
3    Entity : agent;
4    Var_0, Var_1, Var_2, Var_3, Var_4, Var_5, Var_6,
         Var_7, User : param;
5    Image, File: fnode;
6    Tuple, Oldtuple : message;
7    Sessid : cookie;
8    WebNonce : nonce;
9
10   body{
11    while(true){
12     select{
13      % Administrators
14      on(?Entity*->*Actor:http_request(adminlogin, ?
            Var_0.?Var_1, nonec).?WebNonce):{
15      Tuple := Var_0.Var_1.?.?;
16      query(jos_users, Tuple);
17      select{ on(!query(jos_users, Tuple)):{
18       if(Result = ?User.?.?.?){
19       select{ on(isAdmin(User)):{
20        Sessid := fresh();
21        sessions->add((Sessid, usersession, User));
22         Actor*->*Entity:http_response(adminindex,
               Result, Sessid).WebNonce;
23       }}
24       }else{
25         Actor*->*Entity:http_response(adminlogin,
               Result, nonec).WebNonce;
26       }
27      }}
28      }
29
```

```
30      on(?Entity*->*Actor:http_request(adminindex,?Var_0
            ,?Sessid).?WebNonce & sessions->contains((?
            Sessid, usersession, ?User)) & isAdmin(?User) )
            :{
31       Tuple := ?.?.?.?.?.?.Var_0;

32       query(jos_jcalpro_events, Tuple);

33       select{ on(!query(jos_jcalpro_events, Tuple)):{

34          Actor*->*Entity:http_response(adminindex,
                Result, nonec).WebNonce;
35       }}
36      }
37
38     on(?Entity*->*Actor:http_request(adminindex, ?Var_0
            .?Var_1.?Var_2.?Var_3, ?Sessid).?WebNonce &
            sessions->contains((?Sessid, usersession, ?User)
            ) & isAdmin(?User) ):{
39      Tuple := Var_0.Var_1.Var_2.Var_3;

40      insert(jos_users, Tuple);

41      select{on(!insert(jos_users, Tuple)):{

42       Actor*->*Webapplication:htto_response(adminindex,
             Result, nonce).WebNonce;
43      }}
44     }
45      on(?Entity*->*Actor:http_request(adminindex, ?Var_0
            .?Var_1.?Var_2.?Var_3.?Var_4, ?Sessid).?WebNonce
             & sessions->contains((?Sessid, usersession, ?
            User)) & isAdmin(?User) ):{
46      Oldtuple := Var_0.?.?.?;

47      Tuple := Var_1.Var_2.Var_3.Var_4;

48      update(jos_users, Oldtuple, Tuple);

49      select{on(!update(jos_users, Oldtuple, Tuple)):{

50       Actor*->*Webapplication:htto_response(adminindex,
             Result, nonce).WebNonce;
51      }}
52     }
53      on(?Entity*->*Actor:http_request(adminindex,?Var_0
            .?Var_1.?Image.?Var_3.?Var_4.?Var_5.?Var_6, ?
            Sessid).?WebNonce & sessions->contains((?Sessid
            , usersession, ?User)) & isAdmin(?User)):{
```

```
54      Oldtuple := ?.?.?.?.?.?.Var_6;

55      Tuple := Var_0.Var_1.Image.Var_3.Var_4.Var_5.Var_6
            ;
56      update(jos_jcalpro_events, Oldtuple, Tuple);

57      select{ on(!update(jos_jcalpro_events, Oldtuple,
            Tuple)):{
58       writeFile(Image);

59       select{ on(!writeFile(Image)):{

60          Actor*->*Entity:http_response(adminindex,
                Result, nonec).WebNonce;
61       }}
62      }}
63       }
64
65      on(?Entity*->*Actor:http_request(adminindex, ?File
            , ?Sessid).?WebNonce & sessions->contains((?
            Sessid, usersession, ?User)) & isAdmin(?User))
            :{
66      writeFile(File);
67      select{ on(!writeFile(File)):{

68        Actor*->*Entity:http_response(adminindex, none,
              nonec).WebNonce;
69      }}
70      }
71
72      on(?Entity*->*Actor:http_request(adminindex, ?File
            , ?Sessid).?WebNonce & sessions->contains((?
            Sessid, usersession, ?User)) & isAdmin(?User))
            :{
73      readFile(File);
74      select{ on(!readFile(File)):{

75        Actor*->*Entity:http_response(adminindex, Result
              , nonec).WebNonce;
76      }}
77      }
78
79      on(?Entity*->*Actor:http_request(adminindex,none,?
            Sessid).?WebNonce & sessions->contains((?Sessid
            , usersession, ?User)) & isAdmin(?User)):{
80       Actor*->*Entity:http_response(adminindex,fs.db,
             nonec).WebNonce;
81      }
82
```

```
83      % User
84        on(?Entity*->*Actor:http_request(login,?Var_0.?
              Var_1,nonec).?WebNonce):{
85        Tuple := Var_0.Var_1.?.?;

86        query(jos_users, Tuple);

87        select{ on(!query(jos_users, Tuple)):{

88         if(Result = ?User.?.?.?){

89          Sessid := fresh();

90          sessions->add(( Sessid, usersession, User ));

91           Actor*->*Entity:http_response(login,Result,
                 Sessid).WebNonce;
92         }else{
93           Actor*->*Entity:http_response(index,Result,
                 nonec).WebNonce;
94         }
95        }}
96        }
97
98       on(?Entity*->*Actor:http_request(register,?Var_0.?
             Var_1.?Var_2.?Var_3,nonec).?WebNonce):{
99       Tuple := Var_1.Var_3.Var_2.Var_0;

100      insert(jos_users, Tuple);

101      select{ on(!insert(jos_users, Tuple)):{

102        Actor*->*Entity:http_response(register, Result,
             nonec).WebNonce;
103      }}
104      }
105
106      on(?Entity*->*Actor:http_request(addevent,?Var_0.?
             Var_1.?Var_2.?Var_3.?Var_4.?Image,?Sessid).?
             WebNonce & sessions->contains((?Sessid,
             usersession, ?User)) ):{
107     Var_6 := fresh();

108     Tuple := Var_0.Var_1.Image.Var_2.Var_3.Var_4.Var_6;

109     insert(jos_jcalpro_events, Tuple);

110     select{on(!insert(jos_jcalpro_events, Tuple)):{
```

```
111         writeFile(Image);
112         select{ on(!writeFile(Image)):{
113             Actor*->*Entity:http_response(addevent,Result,
                    nonec).WebNonce;
114         }}
115       }}
116       }
117
118         on(?Entity*->*Actor:http_request(index,?Var_0.?
                Var_1.?Var_2.?Var_3,nonec).?WebNonce):{
119         Tuple := Var_0.Var_0.?.Var_1.Var_2.Var_3.?;
120         query(jos_jcalpro_events, Tuple);
121         select{ on(!query(jos_jcalpro_events, Tuple)):{
122           Actor*->*Entity:http_response(index,Result,nonec
                ).WebNonce;
123         }}
124         }
125
126       % Malicious actions
127       on(?Entity *->* Actor : http_request(evil_file,
              none, nonec).?WebNonce & fs->contains(file(
              evil_file))):{
128         Actor -> i : fs;
129       }
130       on(?Entity *->* Actor : http_request(evil_file,
              none, nonec).?WebNonce & fs->contains(file(
              evil_file))):{
131         Actor -> i : db;
132       }
133     }
134   }
135  }
136 }
```

6.3.2.2 The analysis of Cittadiverona

In collaboration with Virtuopolitan S.r.l., I cloned Cittadiverona into a secure environment where I was able to freely test the web application without causing any harm to the real system. I started the analysis by performing a vulnerability assessment by using common vulnerabilities scanners, in particular I used Arachni [5] and OWASP Zed Attack Proxy (ZAP) [62]. This

Table 6.1: Vulnerabilities identified in the Cittadiverona case study

SQLi	Unrestricted file upload	File inclusion	XSS (Reflected)	XSS (Stored)	CSRF
5	2	0	6	2	No protection

assessment allowed to discover a number of common vulnerabilities, which are summarized in Table 6.1.

However, a proper security analysis does not stop after such a vulnerability assessment but continues with a penetration testing phase that aims at understanding how the identified vulnerabilities could be used to harm the web application. This is where WAFEx plays a fundamental role by helping the security analyst in generating AATs where vulnerabilities are combined to violate a security property.

I run WAFEx on the ASLan++ specification to test whether or not the attacker could access the database (Listing 6.5).

Listing 6.5: Confidentiality goal for accessing the database in the Cittadiverona case study

```
[](!(iknows(db)))
```

I was able to generate a total of five AATs, four of which are similar to the AATs already discussed for the Multi-Stage case study. This is not surprising as Multi-Stage shares some basic functionalities (such as user creation, user login, search engine) with Cittadiverona. In fact, Multi-Stage was designed to mimic a real-world web application by implementing the most common functionalities that web applications provide nowadays, and Cittadiverona is a real-world example of how such functionalities are actually being deployed.

However, Cittadiverona is, of course, different from, and much more complex than, Multi-Stage and thus it was possible to generate a new AAT different from the previous ones showing a more complex way to access the database by leveraging functionalities that Cittadiverona implements but Multi-Stage does not.

Abstract Attack Trace #1

This first AAT (Figure 6.7) shows a simple attack where the attacker might be able to exploit a SQLi in the login phase to directly access the entire database. The attacker sends to the web application a request for the page `login` by sending the constant `malicious` and `Password(84)` as login credentials, and the constant `nonec` meaning no cookie is sent (①). The web application, when performing the query for checking the credentials (②), is forced into executing one of the malicious actions I formalized in § 5.5. In this case, the database answers to the login query with the content of the database `{{bob.bobpasswd.bobname.bobsurname.bobphone.bobavatar}}` (③) meaning that the attacker used a SQLi attack to dump the entire database. The

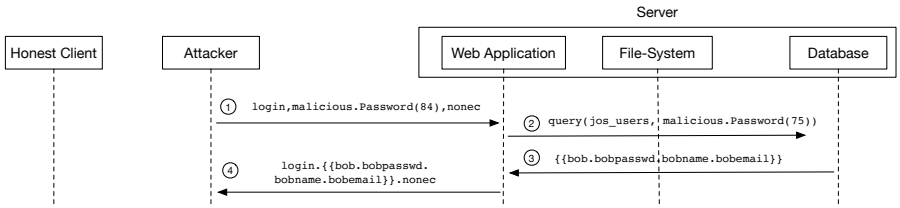


Fig. 6.7: AAT #1 for accessing the database in Cittadiverona

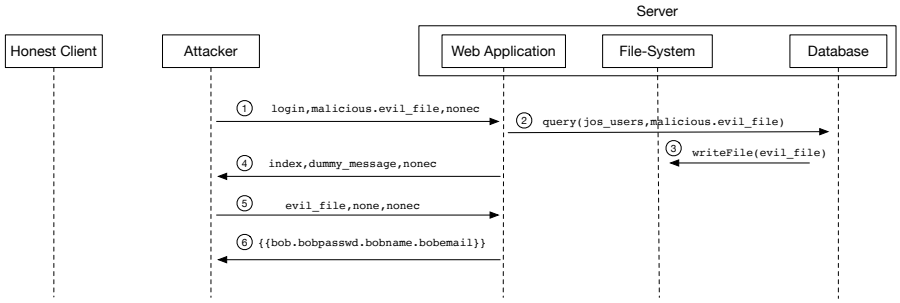


Fig. 6.8: AAT #2 for accessing the database in Cittadiverona

result of executing the query is then sent back to the attacker as a response to the login phase (④).

Abstract Attack Trace #2

I disabled the branch that allows the attacker to force the dump of the entire database and ran the model checker again to generate a different AAT (Figure 6.8). The attacker sends to the web application a request for the page `login` by sending the constant `malicious` and the constant `evil_file` as login credentials (①). The web application performs a query to the database (②) and, because of the constant `malicious`, the database is forced into executing one of the malicious actions I formalized in § 5.5. In this case, the constant `malicious` triggers the possibility of directly writing to the file-system. The file being written is the file represented by the constant `evil_file` (③). Since the write operation does not return any value, the web application simply answers to the attacker with `dummy_message` (④). The attacker can now use the `evil_file` to have direct access to the database (⑤ and ⑥).

Abstract Attack Trace #3

I disabled the branch that allows the attacker to force the database into writing to the file-system and ran the model checker again to generate a different AAT (Figure 6.9). The attacker sends to the web application a request for

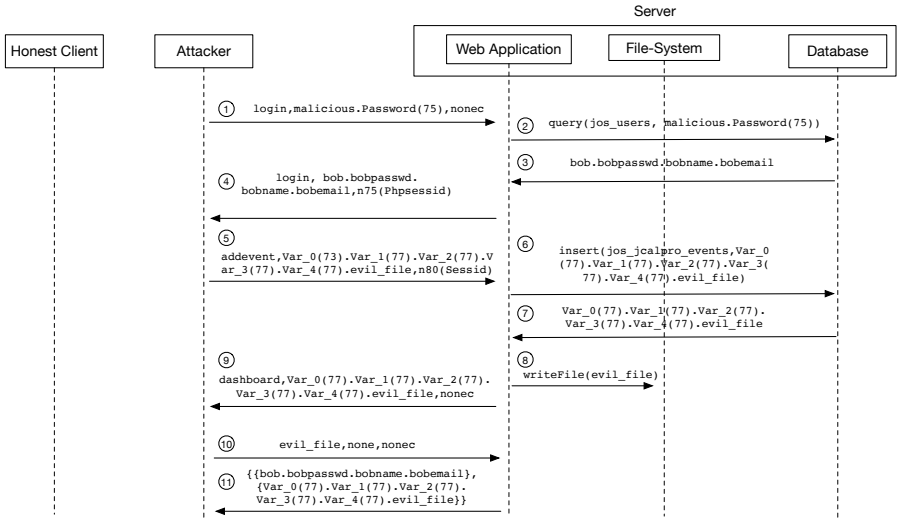


Fig. 6.9: AAT #3 for accessing the database in Cittadiverona

page `login` by sending the constant `malicious` and `Password(75)` as login credentials (①). Once again the database is forced into executing one of the malicious actions I formalized in § 5.5, which in this case would be to bypass the login process and have access without knowing correct credentials (② and ③). The attacker thus gains access as the user `bob` (④), who is a legitimate user in the web application. The attacker can now take advantage of the page `addevent` that gives the possibility to logged in users to create new events in the web application. The attacker exploits this functionality to upload a remote shell as an image for the newly created event (⑤). The request to page `addevent` causes the execution of a query that inserts the details of the new event in table `jos_jcalpro_events` (⑥) and returns a tuple to the web application with the response from the database (⑦). The request to page `addevent` also causes the upload to the file-system of the file `evil_file` (⑧). The web application then answers to the attacker with the details of the newly created event (⑨). Finally, like in the previous AAT, the attacker can now take advantage of `evil_file` (⑩) to have direct access to the database (⑪).

Abstract Attack Trace #4

I generated a fourth, more complex AAT showing how the attacker can have access to the database even when no SQLi attack is possible. I removed the possibility for the attacker to perform any kind of SQLi on the database and generated the AAT in Figure 6.10. In this AAT the interaction starts with the honest client sending a request for page `login` providing credentials for an administrator user (`admin` and `adminpasswd` ①). The web application

performs a query on the database (②) and answers with the details of the administrator (③). The honest client is then granted access to the web application and the session value `n83(Phpsessid)` is sent back to him along with the result of executing the login query (④). The attacker now performs a CSRF attack on the honest client by sending to him a malicious request for page `index` along with the constant `malicious` (⑤). The CSRF attack forces the honest client into performing a request to the page `index` using the constant `malicious` as parameter (⑥). In the Cittadiverona case study, the page `search` is used to search an event in the database, thus the web application performs a query to the database using the constant `malicious` (⑦). The execution of such a query does not produce any tuple, and therefore the constant `none` is returned by the database (⑧). When page `index` answers to a request, it reflects the value of the parameter searched along with the result of the query. The web application then answers to the honest client by sending back the constant `malicious` along with the result of executing the query, which in this case is `none` (⑨). This is where the honest client receives the constant `malicious` and the CSRF attack actually becomes a reflected XSS attack. Recall that based on my formalization (§5.7), whenever the honest client receives the constant `malicious`, he is forced into sending his knowledge to the attacker. The honest client thus sends his knowledge, represented by `{n83(Phpsessid),nonec}`, to the attacker (⑩). The attacker can now use the session values `n83(Phpsessid)` to impersonate the administrator thus performing a session hijacking and have access to administrative functionalities. The attacker does so and performs a request for the backup functionality accessible at page `admininindex` (⑪), which causes the web application to respond with the entire file-system and database of the web application (⑫), thus violating the security property.

Abstract Attack Trace #5

I disabled the possibility for the attacker to perform any SQLi and run the analysis on the model to see if the attacker could find a way to access the entire database. This generated the AAT in Figure 6.11.

The attacker logs in the web application by using credentials `bob` and `bobpasswd` (①), and then the web application performs a query to verify the credentials (② and ③). The web application answers to the login request and generates a new session `n80(Sessid)` that is sent back to the attacker (④). The attacker is now logged in as a normal user and can insert events in the database. He thus uses this functionality to store the constant `malicious` in the database. The attacker performs a request for creating a new event by sending the constant `malicious` as the title for a new event and the session value `n80(Sessid)` that identifies him as a legitimately logged in user (⑤). The web application performs an insert query on the database (⑥) and a subsequent write on the file-system to store the image associated to the newly

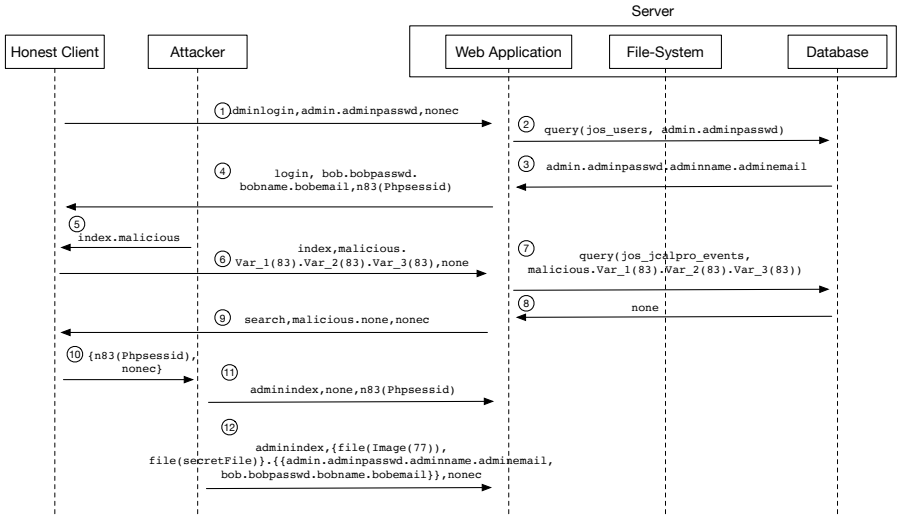


Fig. 6.10: AAT #4 for accessing the database in Cittadiverona

created event (⑦). Finally, the web application answers to the attacker with the details of the newly created event (⑧).

It is worth noting that the message from the web application to the attacker (⑧) contains the same variables sent by the attacker (⑤) representing the new event plus the constant `n82(Var_6)`, which represents the id of the new event. At this point, the honest client logs in the web application as an administrator user, providing credentials `admin` and `adminpasswd` (⑨). The web application performs a query to verify the credentials (⑩ and ⑪) and answers to the honest client with a new session identified by `n74(Sessid)` (⑫). The honest client can now view the events in the database and does so by sending a request to view the event identified by `n82(Var_6)` that was previously created by the attacker (⑬). The web application queries the database (⑭) and retrieves the tuple inserted by the attacker (⑮). The web application then answers to the honest client by sending the result of the query, which contains the constant `malicious` (⑯). The formalization of the honest client (§ 5.7) defines that whenever the honest client receives the constant `malicious`, it is forced into sending his knowledge (which in this case is `{admin,adminpasswd,n74(Sessid),nonec}`) to the attacker (⑰). The attacker thus gains the knowledge of the honest client, which includes the credentials of the administrator `admin`, `adminpasswd` along with the constant `n74(Sessid)` representing a valid and active session for the administrator. This means that the attacker can hijack the administrator's session. The attacker does so and performs a request for the backup functionality (⑱), which

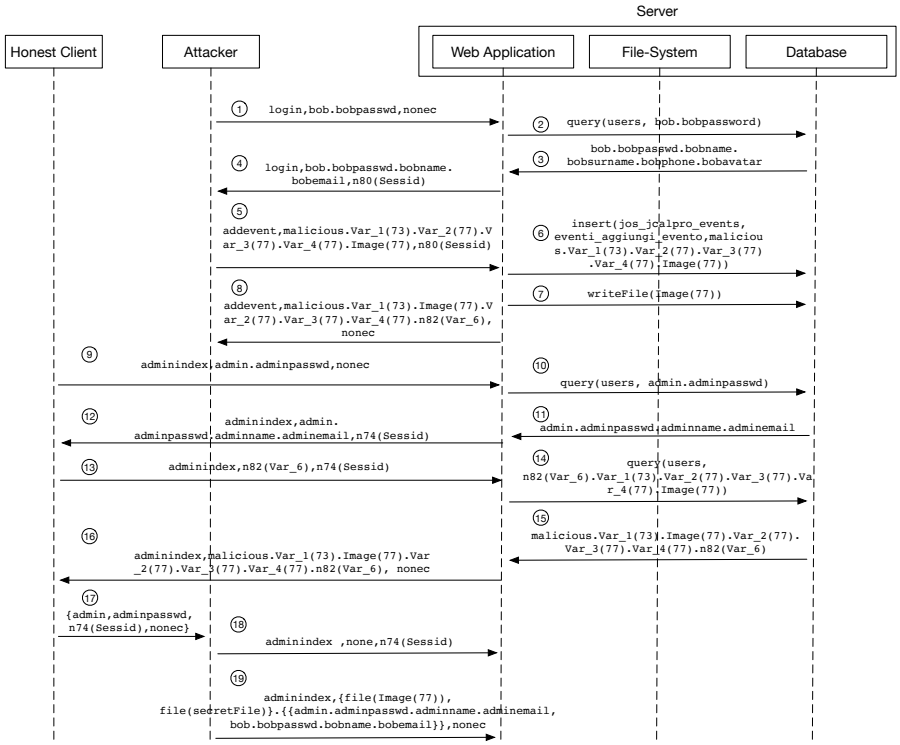


Fig. 6.11: AAT #5 for accessing the database in Cittadiverona

causes the web application to respond with the entire file-system and database of the web application ((19)), thus violating the security property.

This AAT shows the exploitation of a stored XSS in order to perform a session-hijacking attack on the administrator so that the attacker can log in as administrator and access the database via the backup functionality. I believe this AAT to be particularly representative of the strength of possibilities provided by this approach. While the attack might look obvious to the most skilled penetration testers, it is important to remember that, except for WAFEx, *no automatic tool is currently able to identify such an attack* and only a manual investigation can be effective.

6.3.2.3 Concretization

I ran WAFEx on a Mac Book laptop (Intel i5-4288U with 8G RAM and Python3.5). The execution time of the model-checking phase for generating the AAT in Figure 6.11 was around 80s. The AAT was successfully exploited by WAFEx showing that it was indeed possible to perform the attack and thus have access to the entire database.

6.4 Conclusions

In this chapter, I have described WAFEx, a prototype tool that I developed to implement my approach into a model-based testing software to help the security analyst in the exploitation phase of a pentest. I have described how to use the Model Creator plug-in included in WAFEx in order to generate a formal model of a web application and how to use WAFEx for generating and testing AATs. Finally, I have described the application of WAFEx to two real-world case studies: Multi-Stage and Cittadiverona; and shown how WAFEx helped in identifying previously unknown attacks to Cittadiverona.

7

Related work

To the best of my knowledge, this work is the first attempt to show how model-checking techniques and the standard DY attacker model can be used for the generation of attack traces where multiple vulnerabilities are used to violate security properties of web applications. There are, however, a number of previous works that are closely related to what I presented in this thesis and that are thus worth discussing: manual analysis and model-based testing.

7.1 Combination of vulnerability assessment and penetration testing

A combination of vulnerability assessment and penetration testing remains the leading methodology for the security analysis of web applications. The vulnerability assessment phase has received much attention and several different tools have been proposed to help the security analyst in identifying the presence of a vulnerability. On the other hand, penetration testing has typically been left to the experience of the security analyst. This is because the human component is crucial in evaluating the security of the web application. Related to the vulnerability assessment phase, I mention the main, freely available, tools that greatly support the security analyst in finding the presence of vulnerabilities in web applications: sqlmap, sqlninja, DotDotPwn, WFuzz, xss-proxy, Burp Proxy (Community Edition) and OWASP Zed Attack Proxy (ZAP).

sqlmap [74] and sqlninja [75] are tools used to find SQLi entry points. sqlmap supports many different databases, whereas sqlninja only supports the Microsoft SQL server and provides features that are specific to attacking the Microsoft SQL server that sqlmap does not provide. DotDotPwn [28] and WFuzz [31] are used for fuzz testing when searching for injection points. DotDotPwn is specifically tailored to test for directory traversal vulnerabilities, whereas WFuzz is a general purpose fuzzer that given a library of payloads

and one or more injection points, performs many requests to the web application trying every payload in every injection point. On the client side, it is worth to mention the tool xss-proxy [89], which is used for advanced XSS attacks. HTTP/S proxies also play an important role in understanding the behavior of a web application. Specifically, it is worth to mention Burp Proxy (Community Edition) [68] and OWASP Zed Attack Proxy (ZAP) [62].

As I already stated, none of these tools give any clue on how a vulnerability can be used in the web application under analysis and they don't say if an attack that uses that vulnerability can actually be carried out. WAFEx, on the other hand, aims at helping the security analyst understand how vulnerabilities related to web application can violate a given security property, actually helping the penetration testing phase rather than the vulnerability assessment phase.

7.2 Model-based testing

Model-based testing has been steadily maturing into a viable alternative and/or complementary approach. I now describe model-based testing approaches that are closely related to my research.

In [3], Akhawe et al. presented a methodology for modeling web applications and considered five case studies modeled in the Alloy [42] language. The idea is similar to this approach, but they defined three different attacker models that should find web attacks, whereas I have shown how the standard DY attacker can be used. They also represent a number of HTTP details that are not require in the formalization that I propose. Finally, and most importantly, they don't take combination of attacks into consideration.

In [18], Büchler et al. presented SPaCiTE, a formal approach for the security analysis of web applications. SPaCiTE is a model-based security testing tool that starts from a secure ASLan++ specification of a web application and, by mutating the specification, automatically introduces security flaws. SPaCiTE implements a mature concretization phase, but it mainly finds vulnerability entry points and tries to exploit them, whereas the main goal of my methodology is to consider how the exploitation of one or more vulnerabilities can compromise the security of the web application.

The "Chained Attack" approach of [20] considered multiple attacks to compromise a web application. The idea is close to the one I present in this paper. However, the "Chained Attack" approach does not consider file-system vulnerabilities nor interactions between vulnerabilities, which means that with that formalization it would be impossible to represent a SQLi to access the file-system. Finally, the "Chained Attack" approach requires an extra effort of the security analyst, who should provide an instantiation library for the concretization phase, while I relay on well-known external state-of-the-art tools.

In [70], Rocchetto et al. model web applications to search for CSRF attacks. They limit the analysis to CSRF only but their work inspired the intro-

duction of CSRF in my formalization. Still, I did not use their formalization as it is since I wanted to create a more comprehensive representation of the honest client that could be used in every specification.

The methodology proposed by Armando et al. in [9, 7] is focused on binding the specification of security protocols to actual implementations. The methodology starts with the definition of an abstract model (written using a role-based language) of the HTTP messages composing the security protocol. A model checker is then used to derive a counterexample violating some given security properties. The abstract messages, contained in the counterexample, are then mapped to concrete messages used to test the web application. The results differ from mine since WAFEx deals with web applications and vulnerabilities related to them rather than with the security of protocols that are employed by web applications.

In [65], Pellegrino et al. introduce *Deemon*, an automated testing framework to discover CSRF attacks. The approach they propose is based on an analysis of a web application to create a model that uses property graphs to represent information such as execution traces and data flows. Deemon is specifically tailored to finding CSRF and requires access to the source code of the web application, whereas WAFEx can be used without having access to the source code; moreover, I do not focus on finding any specific vulnerability but I rather combine multiple vulnerabilities to violate a security property.

In [13], Backes et al. introduce an analysis technique for PHP applications based on code property graphs. They start from the source code of a PHP web application and create a graph structure representing a canonical representation of the web application incorporating the syntax, control flow, and data dependencies. Finally, they use queries for graph databases to describe web applications vulnerable to specific vulnerabilities. They claim that their approach works with high-level, dynamic scripting languages such as PHP but it is unclear if it can be applied to non scripting and typed languages such as Java servlet. In contrast, I designed WAFEx to be as technology-independent as possible and thus WAFEx is not tied to a specific programming language or environment. Moreover, as already stated for [65] and throughout this thesis, the goal of WAFEx is to find attacks that, combining multiple vulnerabilities, violate a security property of a web application.

8

Summary of contributions

In this thesis I have presented a formal, model-based testing approach for the security analysis of web applications. More specifically, I have proposed the formalization of the most dangerous vulnerabilities of web applications: SQLi, XSS, CSRF and file-system related. To formally represent such vulnerabilities, I have proposed the formalization of four entities representing the vulnerable parties in a web application: the file-system, the database, the web application itself and the honest client. The formalizations of the file-system, the database and the honest client are general enough and can be reused in any scenario. The formalization of the web application, on the other hand, depends on the scenario being modeled and thus it is not possible to provide a single entity that can be reused in every model. However, I have provided a skeleton of a specification and a series of guidelines on how to formally represent a web application. Using this formalization I have shown how the DY attacker can be used in order to find and exploit vulnerabilities of web applications. More specifically, I have shown how the DY attacker can find multi-stage attacks to web applications that, to the best of my knowledge, no other tools can find. Multi-stage attacks involve the combined exploit of multiple vulnerabilities with the aim of finding complex attacks to web applications. The formalization I propose does not search for payloads that can be used to exploit a particular vulnerability, but rather it exploits vulnerabilities of web applications. Moreover, my formalization is independent from the many different technologies used to implement web applications.

I have implemented a plug-in for the Burp Proxy that can be used to create the formal model of a web application in the formal language ASLan++. I have also implemented a prototype tool called WAFEx that takes two inputs: the formal model of a web application written in ASLan++ and a concretization file. WAFEx makes use of model-checking techniques in order to find AATs violating security properties of web applications. Specifically, WAFEx runs the CL-AtSe model checker on the ASLan++ model to generate an AAT. If an AAT is generated, WAFEx automatically tests it against the real web application. Testing an AAT requires the exploitation of one or more vulnera-

bilities which means that WAFEx needs to use appropriate payloads in order to exploit vulnerabilities. Since the goal of this approach is not to find payloads for exploiting vulnerabilities, WAFEx generates such payloads by using state-of-the-art tools, specifically Wfuzz [31] and sqlmap [74].

I have first applied WAFEx to well-known vulnerable web applications (WebGoat, DVWA and Gruyere) to check whether my approach was able to identify the vulnerabilities I was interested in. The results I obtained in this first analysis were very promising so that I took a step further and applied WAFEx to two real-world case studies: Multi-Stage and Cittadiverona. The use of WAFEx on these two case studies resulted in the generation of AATs representing attacks that, to the best of my knowledge, no other tool for the security analysis of web applications is able to identify. The results I have presented in this thesis clearly show that model-checking techniques can be used to identify complex attacks that so far only the manual analysis carried out by a penetration tester could find. The results are promising but further research is required for such an approach to be applied in a professional environment.

9

Future work

The results I have presented in this thesis clearly show that model-checking techniques can be used to identify complex multi-stage attacks to web application that, so far, only the manual analysis carried out by a penetration tester could find. As for any research work, there are several directions that could be deepened to enhance the approach I propose and make it read to be applied in a professional environment. I see two main ares for future work: model creation and AATs generation.

The model creation of a web application is a crucial part in the formal approach I propose. The creation of a model is a difficult task as it might introduce errors on the specification or might result in non-termination of the analysis. The Burp Proxy plug-in that I propose is a fist attempt at automatic model creation but I envision a more elaborate and fully automatic model creation phase which is capable of extracting implementation details such as the interaction between the web application and all the other entities involved in the communication. In particular, useful interaction with my approach might come from adopting automatic model extraction techniques like the one implemented in JModex [50].

Finally, let's consider generation of AATs. As already stated, the CL-AtSe model checker that is currently employed in WAFEx is not capable of generating multiple AATs. This is not an issue in the context of this thesis as it can still be used to show that the approach I propose is actually capable for generating multi-stage attacks. However, a professional environment requires the use of a model checker that is capable of generating multiple AATs. This is why the model-checking phase needs to be improved by adopting an alternative solution or implement multiple AATs generation directly into CL-AtSe. Furthermore, improvements might come from the implementation of new model-checking approaches such as [71] which might allow for a faster analysis.

References

1. Acunetix. https://www.acunetix.com/.
2. Adobe. Flash. https://www.adobe.com/products/flashruntimes.html.
3. Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a Formal Foundation of Web Security. In *CSF*. IEEE, 2010.
4. Apache software foundation. Apache HTTP Server Tutorial: .htaccess files. https://httpd.apache.org/docs/current/howto/htaccess.html.
5. http://www.arachni-scanner.com/.
6. Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, David von Oheimb, Giancarlo Pellegrino, SerenaElisa Ponta, Marco Rocchetto, Michael Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani, and Luca Viganò. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In *TACAS*, LNCS 7214, pages 267–282. Springer, 2012.
7. Alessandro Armando, Roberto Carbone, Luca Compagna, Keqin Li, and Giancarlo Pellegrino. Model-Checking Driven Security Testing of Web-Based Applications. In *3rd International Conference on Software Testing, Verification and Validation, ICST 2010*, pages 361–370. IEEE Computer Society, 2010.
8. Alessandro Armando and Luca Compagna. SATMC: a SAT-based model checker for security protocols. In *JELIA*, LNAI 3229, pages 730–733. Springer, 2004.
9. Alessandro Armando, Giancarlo Pellegrino, Roberto Carbone, Alessio Merlo, and Davide Balzarotti. From model-checking to automated testing of security protocols: Bridging the gap. In *Tests and Proofs - 6th International Conference, TAP 2012*, pages 3–18. Springer, 2012.
10. AVANTSSAR. Deliverable 2.1: Requirements for modelling and ASLan v.1, 2008. www.avantssar.eu.
11. AVANTSSAR. Deliverable 2.2: ASLan v.2 with static service and policy composition., 2009. http://www.avantssar.eu.
12. AVANTSSAR. Deliverable 2.3 (update): ASLan++ specification and tutorial, 2011. http://www.avantssar.eu.
13. Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In

2017 IEEE European Symposium on Security and Privacy (EuroS&P), pages 334–349, April 2017.
14. David Basin, Sebastian Mödersheim, and Luca Vigano. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2005.
15. Matt Bishop. About penetration testing. *IEEE Security & Privacy*, 5(6):84–87, 2007.
16. Abian Blome, Martín Ochoa, Keqin Li, Michele Peroli, and Mohammad Torabi Dashti. VERA: A flexible model-based vulnerability testing tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013*, pages 471–478, 2013.
17. Matthias Büchler. *Semi-Automatic Security Testing of Web Applications with Fault Models and Properties*. PhD thesis, Technical University Munich, 2015.
18. Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Semi-automatic security testing of web applications from a secure model. In *SERE*.
19. Matthias Büchler, Johan Oudinet, and Alexander Pretschner. SPaCiTE – Web Application Testing Engine. In *5th IEEE International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 858–859, 2012.
20. Alberto Calvi and Luca Viganò. An Automated Approach for Testing the Security of Web Applications Against Chained Attacks. In *31st ACM/SIGAPP Symposium on Applied Computing (SAC)*. ACM Press, 2016.
21. Marcus Carey. Penetration Testing vs. Vulnerability Scanning - What's the Difference? https://www.alienvault.com/blogs/security-essentials/penetration-testing-vs-vulnerability-scanning-whats-the-difference.
22. Steve Christey. The 2009 CWE/SANS top 25 most dangerous programming errors. http://cwe.mitre.org/top25.
23. Bernardo Damele and Assumpção Guimarães. Advanced SQL injection to operating system full control. In *BlackHat EU*, 2009.
24. Federico De Meo, Marco Rocchetto, and Luca Viganò. Formal Analysis of Vulnerabilities of Web Applications Based on SQL Injection. In *Security and Trust Management (STM)*, LNCS 9871. Springer, 2016.
25. Federico De Meo and Luca Viganò. A Formal Approach to Exploiting Multi-stage Attacks Based on File-System Vulnerabilities of Web Applications. In *ESSoS*, pages 196–212, 2017.
26. Federico De Meo and Luca Viganò. A Formal and Automated Approach to Exploiting Multi-Stage Attacks of Web Applications. (submitted).
27. Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29, 1983.
28. DotDotPwn - The Directory Traversal Fuzzer. https://github.com/wireghoul/dotdotpwn.
29. Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *DIMVA*, LNCS 6201. Springer, 2010.
30. Damn Vulnerable Web App (DVWA). http://www.dvwa.co.uk.
31. Edge-security. Wfuzz: The Web Bruteforcer. https://github.com/xmendez/wfuzz.
32. Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Security Testing: A Survey. *Advances in Computers*, 101:1–51, 2016.

33. Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler, and Alexander Pretschner. Model-based security testing: A taxonomy and systematic classification. *Software Testing, Verification and Reliability*, 2015.
34. Jeff Forristal. ODBC and MS SQL server 6.5. *Phrack*, 8(54), 1998.
35. Fergal Glynn. Vulnerability Assessment and Penetration Testing. http://www.veracode.com/security/vulnerability-assessment-and-penetration-testing.
36. Greenbone Networks GMBH. Open Vulnerability Assessment System (OpenVAS). http://www.openvas.org/.
37. Shashank Gupta and Brij Bhooshan Gupta. Cross-Site Scripting (XSS) attacks and defense mechanisms: Classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 2015.
38. William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *ISSSE*, 2006.
39. Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Commun. ACM*, 19(8):461–471, 1976.
40. ISECOM. Open Source Security Testing Methodology Manual (OSSTMM). http://www.isecom.org/research/.
41. Daniel Jackson. Alloy Analyzer. http://alloy.mit.edu/alloy/index.html.
42. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.
43. Joomla! https://www.joomla.org.
44. Bruce Leban, Mugdha Bendre, and Parisa Tabriz. Gruyere: Web Application Exploits and Defenses. https://google-gruyere.appspot.com/, 2015.
45. Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-Based Vulnerability Testing for Web Applications. In *SECTEST'13*, pages 445–452. IEEE CS Press, 2013.
46. Rapid7 LLC. Metasploit. https://www.metasploit.com/.
47. Microsoft. ActiveX. https://www.microsoft.com/com/tech/activex.asp.
48. Microsoft. ASP documentation: Including Files in ASP Applications. https://msdn.microsoft.com/en-us/library/ms524876(v=vs.90).aspx.
49. Microsoft. Silverlight. https://www.microsoft.com/silverlight/.
50. Petru Florin Mihancea and Marius Minea. JMODEX: model extraction for verifying security properties of web applications. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pages 450–453, 2014.
51. MITRE. Common Attack Pattern Enumeration and Classification (CAPEC). https://capec.mitre.org/.
52. Mozilla. JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript.
53. Multi-Stage Web Application. https://github.com/rhaidiz/multi-stage.
54. Netsparker Web Application Security Scanner. https://www.netsparker.com/web-vulnerability-scanner/.
55. Oracle. MySQL. https://www.mysql.com.
56. Oracle. MySQL LOAD_FILE. http://dev.mysql.com/doc/refman/5.7/en/string-functions.html#function_load-file.
57. Oracle. The Java EE 5 Tutorial: Reusing Content in JSP Pages. http://docs.oracle.com/javaee/5/tutorial/doc/bnajb.html.

58. OWASP. DirBuster. https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project.
59. OWASP. OWASP File System. https://www.owasp.org/index.php/File_System.
60. OWASP. OWASP Testing Guide v4. https://www.owasp.org/index.php/OWASP_Testing_Project.
61. OWASP. Top 10 for 2013. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.
62. OWASP. Zed Attack Proxy Project (ZAP). https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.
63. OWASP. WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2011.
64. OWASP. Testing Guide v 4.0, 2015.
65. Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1757–1771, 2017.
66. Michele Peroli, Federico De Meo, Luca Viganò, and Davide Guardini. Mobster: A model-based security testing framework for web applications. *Software Testing, Verification and Reliability*. (to appear).
67. PHP documentation: include. http://php.net/manual/it/function.include.php.
68. Postswigger. Burp Proxy, 2014. https://portswigger.net/burp/proxy.html.
69. Andres Riancho. w3af — Web Application Attack and Audit Framework. http://www.arachni-scanner.com/.
70. Marco Rocchetto, Martín Ochoa, and Mohammad Torabi Dashti. Model-Based Detection of CSRF. In *SEC*, IFIP AICT 428. Springer, 2014.
71. Marco Rocchetto, Luca Viganò, and Marco Volpe. An interpolation-based method for the verification of security protocols. *Journal of Computer Security*, pages 463–510, 2017.
72. Amirmohammad Sadeghian, Mazdak Zamani, and Shahidan M. Abdullah. A Taxonomy of SQL Injection Attacks. *ICICM*, pages 269–273, 2014.
73. SANS Institute. Penetration Testing: Assessing Your Overall Security Before Attackers Do. https://www.sans.org/reading-room/whitepapers/analyst/penetration-testing-assessing-security-attackers-34635.
74. sqlmap: Automatic SQL injection and database takeover tool, 2013. http://sqlmap.org.
75. sqlninja: a sql server injection and takeover tool. http://sqlninja.sourceforge.net/.
76. Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws (2nd Edition)*. John Wiley & Sons, Inc., New York, NY, USA, 2011.
77. Chris Sullo and David Lodge. Nikto. http://www.cirt.net/nikto2.
78. Telerik. Fiddler. https://www.telerik.com/fiddler.
79. Trustwave SpiderLabs. Joomla SQL Injection Vulnerability Exploit Results in Full Administrative Access, 2015. https://www.trustwave.com/Resources/SpiderLabs-Blog/Joomla-SQL-Injection-Vulnerability-Exploit-Results-in-Full-Administrative-Access.
80. Mathieu Turuani. The CL-AtSe Protocol Analyser. In *RTA*, LNCS 4098. Springer, 2006.

81. Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.
82. Alexandre Vernotte, Cornel Botea, Bruno Legeard, Arthur Molnar, and Fabien Peureux. Risk-Driven Vulnerability Testing: Results from eHealth Experiments Using Patterns and Model-Based Approach. In *RISK 2015*, pages 93–109. Springer, 2015.
83. Robert Vibhandik and Arijit Kumar Bose. Vulnerability assessment of web applications – a testing approach. In *ICeND*, pages 1–6, 2015.
84. Luca Viganò. The SPaCIoS Project: Secure Provision and Consumption in the Internet of Services. In *Software Testing, Verification and Validation (ICST)*, 2013.
85. David von Oheimb and Sebastian Mödersheim. ASLan++ — a formal security specification language for distributed systems. In *Formal Methods for Components and Objects, FMCO*, LNCS 6957, pages 1–22. Springer, 2010.
86. Web Application Formal Exploiter (WAFEx). https://github.com/rhaidiz/wafex.
87. Web Application Formal Exploiter (WAFEx) Model Creator. https://github.com/rhaidiz/wafex-model-creator.
88. Wfuzz: The Web fuzzer. https://github.com/xmendez/wfuzz.
89. XSS-Proxy. http://xss-proxy.sourceforge.net/.

A

Appendix

A.1 Modeling DVWA, WebGoat and Gruyere

DVWA, WebGoat and Gruyere are state-of-the-art testing environments vulnerable to the most dangerous vulnerabilities of web applications. The purpose of these environments is to provide pentesters with a safe space where they can learn how to exploit the main vulnerabilities afflicting the security of web applications. They are structured in a series of different lessons implementing similar vulnerable functionalities such as login, database search and file-system access. The thing that changes from one environment to the other, is the payload required to exploit a vulnerability. Since my formalization does not consider payloads, meaning that I do not search for a payload that exploits a vulnerability, it is possible to develop a single ASLan++ model representing the functionalities of interested that are shared by all these three testing environments. I now describe an ASLan++ model of a web application that I used to test DVWA, WebGoat and Gruyere. The ASLan++ code is shown in Listing A.1 and represents a web application that provides the following four pages:

- a page for creating a password;
- a page for querying the database;
- a page for uploading a file to the file-system;
- a page that includes and external resource in a web page.

The web application handles a request for the page `insertpwd` with two parameters `Newpwd` and `Confpwd` (line 9). The web application now checks whether `Newpwd` and `Confpwd` are equal (line 10) and, if that is the case, performs an insert operation on table `users` (line 11). The web application now waits for the insert operation to be completed (line 12) and answers by sending the result of executing the insert query (line 13).

The web application handles a request for the page `querydb` with parameter `Query` (line 20) which is used to query the database. The web application performs a query (line 21) and waits for the database to perform the query

(line 22). Finally, the web application answers with variable `Result` representing the result of executing the query (line 23).

The web application handles a request for the page `upload` with parameter `Path` used to upload and write a file to the file-system (line 28). The web application performs a writing operation on the file-system for writing file `Path` (line 29) and waits for the file-system to perform the writing operation (line 30). Finally, since the writing operation on the file-system does not produce an output, the web application simply answers by sending the constant `none` (line 31).

The web application handles a request for the page `include` with parameter `Path` used to read a file from the file-system (line 36). The web application performs a reading operation on the file-system (line 37) and waits for the file-system to perform the reading operation (line 38). Finally, the web application answers by sending the variable `Result` representing the result of executing the reading operation (line 39).

Listing A.1: ASLan++ code representing a web application used to test DVWA, WebGoat and Gruyere

```
1  entity Webapplication(Honest, Actor, Database,
       Filesystem: agent) {
2   symbols
3    Entity : agent;
4    Path, Query, Newpwd, Confpwd : message;
5
6   body{
7    while(true){
8     select{
9        on(?Entity*->*Actor:http_request(insertpwd, ?
             Newpwd.?Confpwd, nonec)):{
10       select{ on(Newpwd = Confpwd):{
11         insert(users, Newpwd);
12         select{ on(!insert(users, Newpwd)):{
13             Actor *->*Entity:http_response(insertpwd,
                   Result, none);
14           }
15         }
16         }
17       }
18        }
19
20      on(?Entity*->*Actor:http_request(querydb, ?Query,
           none)):{
21       query(test, Query);
22       select{ on(!query(test, Query)):{
23           Actor*->*Entity:http_response(querydb, Query.
                 Result, none);
24         }
```

```
25          }
26         }
27
28         on(?Entity*->*Actor:http_request(upload, ?Path,
               none)):{
29          writeFile(Path);
30          select{ on(!writeFile(Path)):{
31              Actor*->*Entity:http_response(upload, none,
                    none);
32            }
33          }
34         }
35
36         on(?Entity*->*Actor:http_request(include, ?Path,
               none)):{
37          readFile(Path);
38          select{ on(!readFile(Path)):{
39              Actor*->*Entity:http_response(include, Result
                    , none);
40            }
41          }
42         }
43        }
44       }
45      }
46     }
```

A.2 Exploiting a vulnerability

DVWA, WebGoat and Gruyere are divided in lessons the purpose of which is to exploit a single vulnerability and thus applying WAFEx to these testing environments results in the generation of a simple MSC. In Figure A.1 is shown a MSC representing the messages involved in the exploitation of a single vulnerability in DVWA, WebGoat and Gruyere. The attacker sends a message to a victim entity `Victim` ①. In my formalization, the attacker communicates only with the web application or the honest client, thus the victim entity is alway only one of these two entities. The message sent by the attacker contains the malicious constant `malicious` which represents any and all payloads for exploiting vulnerabilities of web applications as described in § 5.3. Optionally, a number of HTTP requests and responses may occur after the attacker sent the malicious payload ②, representing the interleaving communication between other entities. Finally, a message from the victim is sent to the attacker with the result of triggering a malicious operation ③.

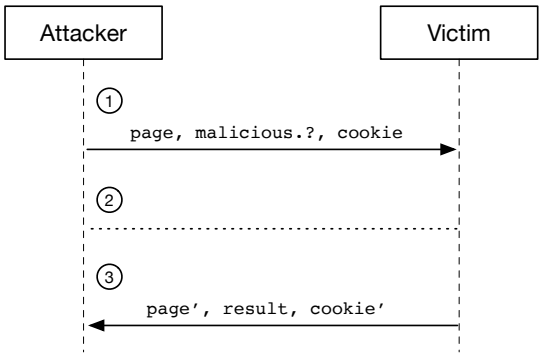


Fig. A.1: Anatomy of exploiting a vulnerability of web applications with WAFEx