University of Verona

DEPARTMENT OF COMPUTER SCIENCE
Graduate School of Science and Engineering

Doctoral Program in Computer Science

S.S.D. INF/01

Cycle XXVII, January 2012 – December 2014

A Model-Based Security Testing Approach for Web Applications

Doctoral Student:
Michele Peroli

Tutor:
Prof. Luca Viganò

Coordinator:
Prof. Paolo Fiorini

Thesis submitted in 2015

Series N°: **TD-05-15**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 - Verona
Italy

Without training, they lacked knowledge.
Without knowledge, they lacked confidence.
Without confidence, they lacked victory.

Julius Caesar

Abstract

Penetration testing is the most common approach for testing the security of web applications, but model-based testing has been steadily maturing into a viable alternative and complementary approach. Penetration testing is very cost-efficient, in the sense that little pen-testing time usually is enough to reveal several bugs, but the experience of the security analyst is crucial; model-based testing relies on formal methods but the security analyst has to first create a suitable model of the application under test.

In this thesis, I propose a formal and flexible model-based framework that supports a security analyst in carrying out security testing of web applications. The main idea underlying this framework is that the use of model-checking techniques can automate the research of possible vulnerable entry points in the web application, i.e., it permits an analyst to perform security testing without missing important checks. Moreover, the framework also allows for reuse: the analyst can collect her expertise into the framework and (re)use it 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. In this way, the potentiality of a single test is not related to the expertise of the single analyst on a specific web application but to the expertise of the entire testing group.

As concrete examples, I consider four case studies in order to show the suitability and flexibility of the framework. Tests for a variety of vulnerabilities has been performed and compared with the ones executed with three benchmark security tools.

Contents

List of Figures

List of Tables

Chapter 1

Overview

1.1 Introduction

1.1.1 Motivation

Since the introduction of the internet, web applications have been constantly increasing their number and the complexity of their services. Nowadays web applications offer services such as banking, shopping, information gathering, web mail, and so on. Web applications security is critical (i) for the users that entrust them with personal information, and (ii) for the companies that provide services (implemented in web applications) as an extension of their business.

On the internet we can find different types of attackers with different technical knowledge, goals and motivations. Known vulnerabilities and attack vectors make simple also for not so expert attackers to jeopardize the security of web applications. Moreover, new vulnerabilities are discovered very often. The research on web applications security tries to limit the exploitation of such vulnerabilities by attackers.

In the literature we can find an abundant number of tools and methodologies for aiding developers and penetration testers to discover and prevent common software security vulnerabilities. At the beginning of "internet security" the majority of techniques were manual, but the need to automate the verification process of web applications has emerged.

Nowadays *penetration testing* [59, 42] is the most common approach for testing the security of web applications. *Model-based testing* [18, 65] is not yet as widespread as penetration testing but it has been steadily maturing into a viable alternative and complementary approach. Both these testing techniques, however, require quite some effort of the security analyst carrying out the tests, even when she may make use of existing tools, guidelines or libraries of common security vulnerabilities and attacks such as [42]. In particular: penetration testing, which ranges from *black-box* to *white-box*, has helped uncover several vulnerabilities, but the experience of the security analyst carrying out the pen-tests is crucial for their success, especially in the case of black-box testing; in model-based testing, a formal model of the web application is used to formally derive test cases, but this requires the security analyst to first create such a model, which may be quite a difficult endeavor especially in the industrial setting.

Testers are also often confronted with situations where existing tools are of little help because (i) they do not account for a particular configuration of the web application and (ii) they do not include tests for certain vulnerabilities.

In this thesis I propose a formal and flexible model-based security testing framework that supports a security analyst in carrying out security testing of web applications. The main focus is on how to reuse the expertise of the penetration testers in a model-based environment with a particular attention on how to simplify the usage of such techniques (i.e., make the framework easily usable also for those analysts who are not accustomed to model-based testing). This is possible through the use of a database of actions and their low-level definition. I have formalized the modeling of web applications, defined the concretization methodology, and discussed four case studies.

From these premises, the main idea is to use the *model checker* in order to find a counterexample (representing which sequence of actions we have to use in order to lead the web application into a probable insecure state) and then use the VERA tool in order to test the application.

1.1.2 Thesis Approach

The framework that I propose (depicted in Figure 1.1, which is described in detail in the next chapters) uses the actions of the *web application* both in the modeling of the application and in the concretization of the test cases.

In the framework, the security analyst creates the *models* of the web applications by defining *actions*; an action, intuitively, is an abstract representation of a part of the web application providing some particular functionality that can be "used"[1] by users through a user interface or a web browser.

Web applications offer various types of functionalities, ranging from general purpose functionalities such as authentication, editing of private information or searching information, to specific functionalities such as reading a newsfeed or purchasing goods from an online shop. During the modeling phase, one is usually not interested in all the implementation details of a single action or in its position in the different pages of a web application in

[1] *Using* an action means that it is possible to perform a sequence of ordered HTTP requests leading the user to access a functionality provided by the action.

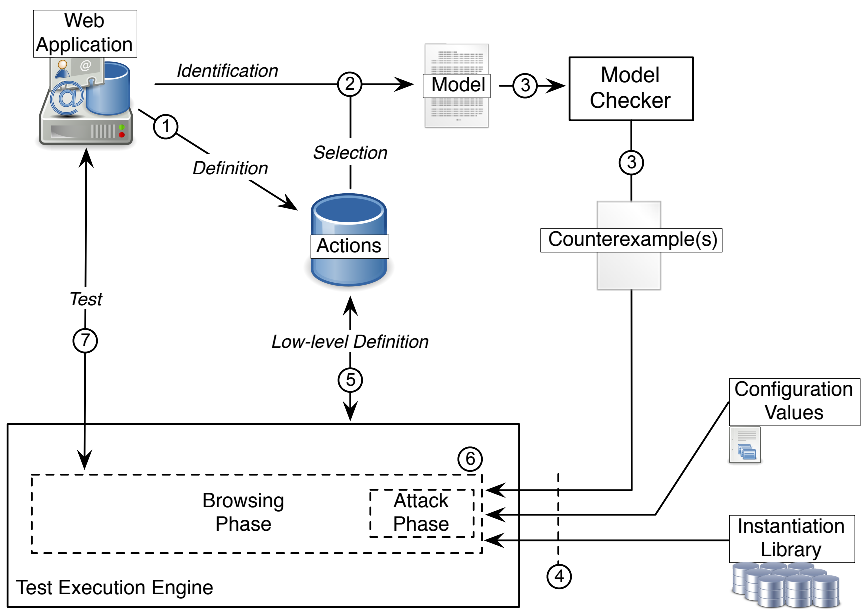


Figure 1.1: The proposed framework for model-based testing of web applications (the arrows refer to interaction between elements or data passed between them, the numbers are used in the text for the explanation of the different phases).

order to consume it; rather, should focus attention on the interaction between actions themselves and between actions and the data they have access to. For example, I specify that the authentication action requires credentials in order to authenticate a user and that administrative actions such as modifying the personal profile can be performed only after the action for authentication (i.e., the editing profile action comes after the authentication action). In the model the actions are decorated with (i) information describing the data they consume (e.g., credentials, text, id, etc.), (ii) the properties describing their behavior (e.g., read from a database, write to a database, etc.), and (iii) security-related information.

After using the actions in order to model the application, the model itself is enriched with a security goal (that the web application must respect during its execution) and it is then fed into a *model checker*, which will return *Counterexamples (CEs)* if any, i.e., 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. The framework thus provides for a *Test Execution Engine (TEE)* that translates a CE in a corresponding sequence of *HTTP requests* that can be performed on the web application.

As will become evident below, actions are simple to identify and can be easily reused. Hence, the presence of actions in my model-based testing framework brings along a number of advantages, including

- *simplicity* (e.g., of the modeling activity),
- *scalability* (the security analyst can identify existing functionalities from a set of known actions or can define new ones to improve the set of actions), and
- *reusability* (in the framework, actions are associated with data they handle, properties they have, and thus reusing actions also means to reuse these associations during the testing of different applications).

Actions are defined by the security analyst who decides the abstraction level and the granularity she 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 creation), consider the "login" functionality. Web applications usually use one of the following types of authentication mechanism (I also add 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 attack and man-in-the-middle attacks are possible.
- *HTTP Client Certificate Authentication* - on the client machine, an attacker can use a legitimate authenticated session in order to attack the web application.
- *Form-based Authentication* - injection vulnerabilities (e.g., Cross-Site Scripting XSS [46], SQL-Injection [51], 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 types of authentication, a security analyst can create an associated action and also (re)use it later to test different models. In fact, this simple example shows how, for a security analyst modeling the web application, it should be *simple* to create an action for the different web application functionalities (e.g., the authentication mechanism), or *scale* or *reuse* some existing action (e.g., she can modify/extend a "Form-based Authentication" action if the web application uses some particular data or has different properties, or she can take an action for authentication from the database). For concreteness, in the examples and the case studies that I consider in Chapter 5, I will use the form-based authentication.

1.1.3 Contributions

As I stated above, the main contribution of this thesis is the proposal of a formal and flexible model-based framework that supports a security analyst in carrying out security testing of web applications. I start by defining a suitable methodology for modeling web applications for security testing (i.e., I define a transition system for web applications that is used to derive test cases for known vulnerabilities). The definition of such methodology is used in order to define models that have to satisfy security goals that define known vulnerabilities to be tested. For concreteness the Alloy analyzer [1] is used as model checker, and the statements that compose its models are used in order to define the transition system. The direct consequence in using Alloy is that I define a standardized structure for web applications' models that can help a security analyst in defining her models.

Since the attack traces derived in the model checking phase are abstract, in order to fill the abstraction gap between them and the implementation of the web application I introduce a suitable concretization methodology, and I present a preliminary version of the implementation of the framework. As an initial proof of concept I show the application of the framework to four case studies:

- WebGoat - A general model and various minimal (in the number of functionalities) models are used in order to show (i) how the model checking phase can be used to derive the attack traces, and (ii) the testing capabilities of the framework.
- Gruyere - Its model (composed by a variety of functionalities) is used in order to (i) test different vulnerabilities, and (ii) show the versatility of the framework in testing these vulnerabilities.
- DVWA - This case study is not interesting for the model checking phase but only for proving the testing capabilities of the framework.
- OnlineShop - It is a fictitious case study that is used in order to derive counterexamples for logic flaws.

As an alternative test execution engine I propose the VERA tool. It has beed developed during the SPaCIoS project [56], I mainly worked on the graphical user interface. Since the VERA tool allows testers to define low-level attacker models I discuss its suitability in becoming the test execution engine of the proposed framework.

1.1.4 Synopsis

This thesis is organized as follows:

- In Chapter 2, I give an overview of different approaches for testing a web application and the vulnerabilities that are tested.
- In Chapter 3, I briefly present the phases of the proposed framework and I show a methodology for modeling web applications for security testing. The goal is to define a transition system for web applications that is used to derive test cases for known vulnerabilities.
- In Chapter 4, I discuss how the transition system introduced in is defined as an Alloy model, how security goals are defined, the concretization methodology and an initial implementation of the framework.
- In Chapter 5, I discuss the application of the framework to the four case studies.
- In Chapter 6, I discuss the related work.
- In Chapter 7, I summarize the contributions of the thesis and discuss future work.

Chapter 2

Preliminaries

There exist several methodologies used by different tools for testing the security of web applications. In the following sections, I give an overview of three approaches that these tools use, and I categorize them based on the knowledge needed for their use:

- In *black-box testing*, the knowledge of the application source code is not needed.
- In *source code analysis*, the knowledge of the source code of the web application is required.
- In *model-based testing*, the knowledge required can be a combination of the previous approaches.

I discuss these three approaches in Sections 2.1 and 2.2. In Section 2.4, I discuss the vulnerabilities that are in the scope of this thesis and are used in the case studies.

2.1 Penetration Testing

Penetration testing [59, 42] is the most common approach for testing the security of web applications, i.e., for generating a set of test cases (namely, pairs of inputs and expected outputs) and executing them on the web application under test in order to acquire more confidence about the secure behavior of the application's implementation or to discover failures (i.e., unexpected behaviors).

2.1.1 Black-box testing (functional testing)

I use the term *black box testing* [59, 42] for describing test methods that are not based directly on application architecture source code. This term connotes a situation in which either the tester does not have access to the source code or the details of the source code are irrelevant to the properties being tested.

In this scenario the tester acquires information by testing the system using test cases (i.e., pairs of input and the expected output). All the tests are carried out from the from the point of view of a user (the external visible behavior of the software), for example, they might be based on requirements, protocol specifications, APIs, or even attempted attacks.

This kind of tests simulates the process of a real hacker but they are time-consuming and expensive.

2.1.2 Source Code Analysis and White-box Testing

Source code analysis (see, e.g., [16]) is the process of checking source code for coding problems based on a fixed set of patterns or rules that might indicate possible security vulnerabilities. This process is a subset of white-box testing (see, e.g., [29]), which is a method of testing applications by checking the structure starting from the source code.

These testing techniques are interesting and useful but I mention them in this section only for completeness as they have not (yet) been the focus of my investigation.

2.2 Model-Based Testing

Model-Based Testing (MBT) [18, 65] consists in a variant of software testing in which a model of the application under testing (web application) is used to derive test cases for its implementation. In contrast to other testing approaches that do not rely on an abstract model of the web application the advantages of adopting MBT are many-fold and mainly related to the involvement of model checkers in the testing process. The most important advantage is the possibility to generate test cases having a specific purpose in an automated way, thanks to the capability of the model checker to provide attack traces. It is indeed possible to formalize the purpose of a test suite in terms of goals and use them, with the model of the web application, in order to cast the test case generation problem as a model checking problem. For example, in the context of coverage testing, one can generate abstract tests by using goals checking the execution of transitions. By doing so, the model checker will provide every execution trace including such transition.

In general, MBT covers three majors tasks:

- automatic generation of abstract test cases from models,
- concretization of abstract tests in order to obtain executable tests, and

- their execution on the web application by using manual or automatic means.

Model-based testing is not yet as widespread as penetration testing but it has been steadily maturing into a viable alternative/complementary approach.

2.3 Types of Tools

In the previous sections we have seen which methods are used by tools in order to test web applications. In this section, we will consider what kind of information are retrieved by the tools and what kind of tests are performed.

Following the partition from [66], I will discuss the following categories (for each one I will show some tools as examples):

- Port scanners,
- vulnerability scanners,
- application scanners,
- web application assessment proxy, and
- packet sniffers.

This classification is not exclusive (some tools can be in more than one category) but it covers all the security tools nowadays available.

Port Scanners

Port scanning tools are used to gather information on the target of the test. Specifically, port scanners attempt to locate which network services are available on each target host. They do this by probing each of the designated (or default) network ports or services on the target system.

Most such tools can also target a specified list of ports and can be configured for setting the speed and ports sequence that they have to scan. Additionally, most port scanners are able to perform a variety of different port probes. They can have the ability to deduce the operating system type and often the version number based on watching the empirical behavior that it exhibits when probed with variations of TCP flag settings.

Vulnerability Scanners

The primary distinction between a port scanner and a network-based vulnerability scanner is that vulnerability scanners attempt to exercise (known) vulnerabilities on their targeted systems, whereas port scanners only produce an inventory of available services.

Vulnerability scanners provide an essential means of meticulously probing each and every available network service on the targeted hosts. Vulnerability scanners work from a database of documented network service security vulnerabilities, exercising each defect on each available service of the target range of hosts.

Traditional vulnerability scanners are generally able to scan only target operating systems and network infrastructure components, as well as any other TCP/IP device on a network, for operating system level weaknesses. They are not able to probe general purpose applications, as they lack any sort of knowledge base of how an unknown application functions.

Some vulnerability scanners are able to attempt to exploit network trust relationships by recursively scanning the targeted network on each compromisable host.

Host-based vulnerability scanners scan a host operating system for known weaknesses and un-patched software, as well as for such configuration problems as file access control and user permission management defects. Although they do not analyze application software directly, they are useful at finding mistakes made in access control, configuration management, and other configuration attributes, even at an application layer.

Application Scanners

Taking the concept of a network-based vulnerability scanner one step further, application scanners began appearing several years ago. These probe general purpose web-based applications by attempting a variety of common and known attacks on each targeted application and page of each application.

Most application scanners can observe the normative functional behavior of an application and then attempt a sequence of common attacks against the application. The attacks include buffer overruns, cookie manipulation, SQL insertion, cross-site scripting (XSS), and the like.

Since the testing is still performed in an entirely black box manner, the utility of such tools is less than any serious testing process.

That is, although failing any of the tests is demonstrably a bad situation, passing all of the tests can only provide, at best, a misplaced sense of security.

Web Application Assessment Proxy

Although they only work on web applications, web application assessment proxies are perhaps the most useful of the vulnerability assessment tools listed here. Assessment proxies work by interposing themselves between the tester's web browser and the target web server. Further, they allow the tester to view and manipulate any and all data content flowing between the two. This gives the tester a great deal of flexibility in trying different "tricks" to exercise application weaknesses in the application's user interface and associated components. This level of flexibility is why assessment proxies are considered essential tools for all black box testing of web applications.

For example, the tester can view all cookies, hidden HTML fields, and other data in use by a web application and attempt to manipulate their values to trick the application into allowing access where the tester should not be able to get to. Changing cookie values such as "customerID" can have startling results on poorly developed applications.

Packet sniffer

Packet sniffers are commonly used to intercept and log traffic passing over a digital network or part of a network. The sniffer captures every packet and, if needed, decodes it showing the values of various fields in the packets.

The captured information is decoded from raw digital form into a human-readable format that permits users of the packet sniffer to easily review the exchanged information. Packet sniffers vary in their abilities to display data in multiple views, automatically detect errors, determine the root causes of errors, generate timing diagrams, etc.

Packet sniffers can also be hardware based, either in probe format, or as is increasingly more common combined with a disk array. These devices record packets (or a slice of the packet) to a disk array. This allows historical forensic analysis of packets without the user having to recreate any fault.

2.3.1 Tools Survey

Following the initial categorization given above, I have collected some tools:

- Some port scanners are:
  - Nmap [33]
  - Scapy [11]
- Some vulnerability scanners are:
  - Tenable Nessus [37]
  - Core Impact [63].
  - Qualys's QualysGuard [53].
  - ISS's Internet Scanner [53].
  - Nikto [60].
  - Wikto [32].
  - Maltego [34].

- Some application scanners are:
  - WebInspect [23].
  - Rational Appscan [27].
  - N-STEALTH [36].
  - Metasploit [54].
  - Canvas [28].
  - Acunetix free ed wvs [2].
  - Hailstorm [25].
  - Beef [10].
  - Wapiti [61].
  - OWASP's lapse [30].
- Some web application assessment proxy are:
  - Paros Proxy [17].
  - Zed Attack Proxy (ZAP) [40].
  - OWASP's WebScarab [39].
  - Burp suite [31].
  - Grendel-Scan [24].
  - PAROS pro desktop [17].
  - Selenium [55].
- Some packet sniffers are:
  - Ettercap [21].
  - Firesheep [22].
  - Wireshark [67].

These tools are quite different and some of them cost money (free limited/demo/trial versions sometimes are available). We can see that the majority of tools belongs to the "Application scanners" category: a first way to read this data is the fact that tools of this kind are very interesting for both the white hat and black hat communities.[1]

After this generic distinction I will go a little bit into details. In Table 2.1 I have categorized tools listed before with respect to the ability of performing some kind of actions.

From the data collected in Table 2.1, together with data from the categorization of the tools, some interesting facts arise:

[1]White hat hackers break security for non-malicious reasons (e.g., while working for a security company) while black hat hackers violate security for malicious reasons (e.g., personal gain).

Table 2.1: Some tools categorized into: A - Scripting / API capabilities, B - handle a model, C - perform attacks.

	A	**B**	**C**
Nmap	X		
Scapy	X		X
Tenable Nessus			X
Core Impact			
Qualys's QualysGuard	X		
ISS's Internet Scanner			
Nikto			
Wikto			
Maltego			
WebInspect	X		X
Rational Appscan	X		
N-STEALTH	X		
Metasploit	X		X
Canvas			X
Acunetix free ed wvs			
Hailstorm			
Beef	X		
Wapiti			
OWAPS Lapse			
Paros Proxy			
Zed Attack Proxy (ZAP)	X		X
OWASP's WebScarab	X		X
Burpsuite			X
Grendel-Scan			
PAROS pro desktop			
Selenium	X		X
Ettercap	X		X
Firesheep	X		X
Wireshark			

- Tools have different goals and capability.
- Some features cross the boundaries of the categorization.
- The most commonly used tools cannot handle a model of a web application.
- There is a balance between tools that can or cannot perform attacks.

2.4 Vulnerabilities

In this section, I introduce the vulnerabilities that are in the scope of this thesis and that will be tested in the case studies in Chapter 5.

2.4.1 Access Control Flaws

Access control enforces the correct decisions about whether a request to access a content or a function from a specific user has to be granted or not. Access controls have to be implemented over functionalities and data. Some of the factors that can make difficult the correct implementation of access controls are:

- The web applications could support numerous user roles with specific privileges for each role.
- Different users could have access to a subset of the total data held within the application.
- The access to specific functions could be granted on the base of the users' identity.

For a given web application, the model for access control is tied to the context in which the application works, and the functionalities that it provides.

Access controls can be divided into three broad categories: vertical, horizontal, and context-dependent. In the following I give a brief explanation for each category.

Vertical access controls allow one to differentiate the users by their roles and give them specific functionalities on the base of their role. For example, a simple differentiation can be between administrative and normal users, where the administrators can access all the normal functionalities plus the ones that permit them to manage the application or the server where it is hosted. Of course more complex examples of vertical access controls can be presented, for example we can use the roles used in WebGoat (Section 5.1): *admin, manager, employee* and *human resources*, and granting to these roles access to specific functions, or give to a user a combination of different roles. *Vertical privilege escalation* occurs when a user can perform functions that his assigned role does not permit him to.

Horizontal access controls allow users to access a certain subset of a wider range of resources of the same type. Enforcing a horizontal access control corresponds to permitting the access to a certain data only to the owner and not to all the users of a system. For example, let A and B be two users of a web application and both have an account that contains private information; the web application should enforcing controls in order to make A not able to access B's account (and vice versa). *Horizontal privilege* *escalation* occurs when a user can view or modify resources to which he is not authorized.

Context-dependent access controls ensure that users' access is restricted to what is permitted given the current application state. Examples of context-dependent access controls can be: only one coupon can be used per transaction, transactions in excess of $2000 has to be reviewed by a person, etc. *Business logic exploitation* occurs when a user can exploit a flaw in the application's state machine to gain access to a key resource. In other words, business logic vulnerabilities are ways of using the legitimate processing flow of an application in a way that results in a negative consequence to the organization.

Access controls are broken [59, 43] if any user (or an attacker) can gain unauthorized access to functionality or data for which he is not authorized. In the following, I give some specific access control issues that can be used to bypass or exclude these controls. These attacks can be used against one or multiple access controls and have to be used accordingly to the controls that the tester (or an attacker) wants to target.

In [43], some specific access control issues are discussed:

- Insecure Id's,
- Forced Browsing Past Access Control Checks,
- Path Traversal, and
- File Permissions.

In the following, I discuss briefly each one of these issues and give some example of possible attacks.

Insecure ID's

Most web sites use some form of ID, key, or index as a way to reference users, roles, contents, objects, or functions. If an attacker can guess these IDs, and the supplied values are not validated to ensure they are authorized for the current user, the attacker can exercise the access control freely to see what they can access. Web applications should not rely on the secrecy of any IDs for protection.

Even if this methodology of attack can affect the security of a web application in a variety of ways, in Section 4.2.1 I will use it in order to test possible horizontal privilege escalations.

Forced Browsing Past Access Control Checks

Many web sites require users to pass certain checks before being granted access to certain URLs that are typically "deeper" down in the site. These checks must not be bypassable by a user that simply skips over the page with the security check.

An attacker can search for unlinked contents in the domain directory, such as temporary directories and files, and old backup and configuration files. These resources may store sensitive information about web applications and operational systems, such as source code, credentials, internal network addressing, and so on, thus being considered a valuable resource for intruders.

Forced browsing [47] is an attack where the aim is to enumerate and access resources that are not referenced by the web application, but are still accessible.

Example 1 (Forced browsing)**.** A scanning tool, like Nikto [60], has the ability to search for existing files and directories based on a database of well-know resources, such as:

```
/system/
/password/
/logs/
/admin/
/test/
```

When the tool receives an HTTP 200 message it means that such resource was found and should be manually inspected for valuable information. △

File Permissions

Many web and application servers rely on *access control lists* provided by the file system of the underlying platform. Even if almost all data are stored on backend servers, there are always files stored locally on the web and application server that should not be publicly accessible, particularly configuration files, default files, and scripts that are installed on most web and application servers. Only files that are specifically intended to be presented to web users should be marked as readable using the OS's permissions mechanism, most directories should not be readable, and very few files, if any, should be marked executable.

Path traversal

Web applications often restrict user access to a specific portion of the file system. These portion of file systems contain files and directories that are (i) intended for user access, and (ii) are necessary to the web application's functionalities.

Path traversal attacks [50] aim to access files and directories that are stored outside the web root folder. By browsing the application, the attacker looks for absolute links to files stored on the web server. By manipulating variables that reference files with "dot-dot-slash (../)" sequences and its variations, it may be possible to access arbitrary files and directories stored on file systems, including application source code, configuration and critical system files, limited by system operational access control. The attacker uses "`../`" sequences to move up to root directory, thus permitting navigation through the file system.

Since browsers accept different encodings, a security analyst has the possibility of encode the "`../`" sequence. As an example, `%2e%2e%2f` represents `../` and `%2e%2e%5c` represents `..\`. These two possible encodings are introduced because the encoding of addresses of files and directories on the file system is OS specific:[2]

	Root directory	Directory separator
Unix	/	/
Windows	<partition>:\	/ or \

Example 2 (Path traversal)**.** The following examples show how an application deals with the resources in use:

```
http://site.com/get-files.jsp?file=report.pdf
http://site.com/get-page.php?home=aaa.html
http://site.com/some-page.asp?page=index.html
```

In these examples, it is possible to insert a malicious string as the variable parameter to access files located outside the web public directory.

```
http://site.com/get-files?file=../../../some dir/some file
http://site.com/../../../some dir/some file
```

The following URLs show examples of unix-like password file exploitation.

```
http://site.com/../../../../etc/shadow
http://site.com/get-files?file=/etc/passwd
```

△

2.4.2 Injection Flaws

Injection flaws [48] allow attackers to relay malicious code through a web application to another system. These attacks include calls to the operating system via system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL. Whole scripts written in perl, python, and other languages can be injected into poorly designed web applications and executed. Any time a web application uses an interpreter of any type there is a danger of an injection attack.

Injection attacks can be very easy to discover and exploit, but they can also be extremely obscure. The consequences can also run the entire range

[2]In a windows system an attacker can navigate only in a partition that locates web root while in the Linux he can navigate in the whole disk.

of severity, from trivial to complete system compromise or destruction. In any case, the use of external calls is quite widespread, so the likelihood of a web application having an injection flaw should be considered high.

Cross-Site Scripting

Cross-Site Scripting (XSS) [46] attacks are a type of injection in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses an input from a user (without validating or encoding it) within the output (that the web application generates).

The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.

Stored XSS Attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim retrieves the malicious script from the server when he requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

Reflected XSS Attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS.

SQL-Injection

A SQL-Injection (where SQL stands for "Structured Query Language") attack [51] consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL-Injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete functions), execute administration operations on the database, such as shutdown the DataBase Management System (DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL-Injection attacks are a type of injection attack in which SQL commands are injected into an input in order to effect the execution of predefined SQL commands.

There are various type of SQL-Injection attacks:

- *SQL-Injection* - where the payloads are used in order to gain access to data or modify them, and
- *Blind SQL-Injection* - where the attack aims to infer data from the asking the database true or false questions and determines the answer based on the applications response.

The latter category will not be discussed in this thesis.

Command-Injection

Command-Injection [45] is an attack in which the goal is to execute arbitrary commands on the host operating system via a vulnerable web application. Command-Injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command-Injection attacks are possible largely due to insufficient input validation.

This attack differs from *Code-Injection* in that Code-Injection allows the attacker to add his own code that is then executed by the application. In Code-Injection attacks, the attacker extends the default functionalities of the web application without the necessity of executing system commands.

2.4.3 AJAX Flaw

Asynchronous Javascript and XML (AJAX) is one of the latest techniques used by web application developers to provide a user experience similar to that of a traditional application. One of the main features of AJAX is that it permits a web application to retrieve data from or to a server asynchronously.

Since AJAX is still a new technology, there are many security issues that have not yet been fully researched. Some of the security issues in AJAX include:

- Increased attack surface with many more inputs to secure than the web applications that does not use AJAX functionalities.
- Exposed internal functions of the web application.
- Client access to third-party resources with no built-in security and encoding mechanisms.
- Failure to protect authentication information and sessions.

- Blurred line between client-side and server-side code, possibly resulting in security mistakes.

A brief introduction to AJAX attacks can be found in [52]. Among the attacks that a security analyst should test for AJAX functionalities, it is worth to mention:

- SQL-Injection,
- Cross-Site Scripting (XSS),
- Client-Side Injection Threats (e.g., XSS-, XML-, DOM-Injection),
- Cross-Site Request Forgery (CSRF), and
- Denial of Service.

2.4.4 Other Vulnerabilities

Brute Force

A brute force attack [44] can manifest itself in many different ways, but primarily consists in an attacker configuring predetermined values, making requests to a server using those values, and then analyzing the responses. For the sake of efficiency, an attacker may use a dictionary attack (with or without mutations) or a traditional brute-force attack (with given classes of characters, e.g.: alphanumerical, special, case (in)sensitive). Considering a given method, number of tries, efficiency of the system which conducts the attack, and estimated efficiency of the system which is attacked, the attacker is able to calculate approximately how long it will take to submit all chosen predetermined values.

Brute-force attacks are often used for attacking authentication and discovering hidden content/pages within a web application. These attacks are usually sent via GET and POST requests to the server. In regards to authentication, brute force attacks are often mounted when an account lockout policy in not in place.

In the following chapters, I discuss the methodology used for modeling web applications, the implementation of the framework, and the application of this methodology to four case studies.

Chapter 3

Modeling Web Applications for Security Testing

In this chapter, I show a methodology for modeling web applications for security testing. The goal is to define a transition system for web applications that is used to derive test cases for known vulnerabilities.

In order to model web application for security testing, I first discuss in details the phases of the framework (Section 3.1). I discuss in Section 3.2 the approach (i.e., what is a transition system, how I chose to define it and what is a model), in Sections 3.3, 3.4 and 3.5 I present a discussion on (i) how I chose to model the users of web applications regarding their knowledge and the data they handle, (ii) the behavior that web applications can manifest through the interaction with a user, and (iii) how to define in the models those information that can be used in order to perform security testing; I complete the modeling approach by defining the states of the transition system (Section 3.6), and the actions that permit its evolution (Section 3.7).

3.1 Phases of the Framework

In this section, I give an overview of the phases of the framework (depicted in Figure 1.1) that will be discussed the following sections and chapters.

The first thing that a security analyst has to do when using my framework is to define actions from the web application (phase ① in Figure 1.1). During this phase, the security analyst has also to check, manually, if some of the existing actions in the database can be (re)used in order to model the web application (if two applications share the same functionality, then she can reuse the associated action), and, if not, she has to insert into the database the new action(s) (see Section 3.7 for further information about the definition of actions).

With the database populated with a proper set of actions, the security analyst has the means to create the *model* of the web application (②). The model includes a selected subset of the defined actions (identified with respect to the web application to be tested), the relation between them, and a specification of the security goal to be tested.

The model is then passed to a *model checker* (③). The framework is general and thus it is not bound to a specific model-checking tool; for concreteness, I employ the Alloy Analyzer [1], which takes a model, its constraints and its security goal written in the Alloy syntax, checks the goal in the model and generates one or more CEs if the goal is violated, i.e., a counterexample shows for what instances of the system and for what actions the security goal does not hold.

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

First, the counterexamples are at a level of abstraction that does not permit us to directly test them on the web application, since it specifies the actions used to violate the goal but not how these actions should be used in the real implementation. The framework thus provides for a *concretization phase*, which relies on the fact that I can define for each action a sequence of HTTP requests to perform on the web application.

In the implementation of the framework, the definition of the relations between actions and HTTP requests (⑤) is performed during the execution of the test cases by a python engine (see Section 4.4 Page 94 for further details).

Second, the counterexamples specify which actions have to be used, but their level of abstraction does not allow for the specification of attack-dependent data. If I have to use a specific payload, the *Instantiation Library* (InstLib) provides it. The InstLib contains data such as attack strings (e.g., payloads for XSS), common malicious input (e.g., a set of passwords for a brute force attack) and scripts to be used as test patterns (i.e., script to be executed client-side in order to test the web application).

The final phase of the framework uses a TEE, an automatic test execution technology that the security analyst can use in order execute the test cases on the web application. The TEE provides a connection with the InstLib and the data, contained in the *Configuration Values* (ConfVal), needed for the interaction with the web application (④), and takes care of "translating" (via the *Low-level Definition* ⑤) the counterexample(s) (④) into executable test cases. At the end of this phase, the information contained in the counterexample(s), the actions and the HTTP requests are thus combined in the creation of a suite of test cases (⑥) that are run on the web application (⑦).

3.2 Modeling Web Applications via a Transition System

In [9], the authors introduce a transition system (TS) as a model to describe the behavior of systems. *TSs* are basically directed graphs where nodes represent states, and edges model transitions, i.e., state changes.

Definition 1 (**Transition System**). *A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, \mathscr{L})$ where*

- *S is a set of states (i.e., some information about a system at a certain moment of its behavior),*
- *Act is a set of actions (whose names are used to describe informally what is happening during a transition),*
- *$\rightarrow \subset S \times Act \times S$ is a transition relation (i.e., how the system can evolve from one state to another),*
- *I is a set of initial states,*
- *AP is a set of atomic propositions (that intuitively express simple known facts about the states), and*
- *$\mathscr{L} : S \rightarrow 2^{AP}$ is a labeling function.*

TS is called finite if S, Act, and AP are finite.

Once a transition system is defined (as will be shown in this chapter), it can be used in order to derive *executions* (i.e., the result of resolving the possible nondeterminism in the system). An execution describes a possible behavior of the transition system. Formally:

Definition 2 (**Execution Fragment**). *Let $TS = (S, Act, \rightarrow, I, AP, \mathscr{L})$ be a transition system. A* finite *execution fragment ϱ of TS is an alternating sequence of states and actions ending with a state*

$$\varrho = s_0 \alpha_1 s_1 \alpha_2 \ldots \alpha_n s_n \quad \textit{such that} \quad s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \quad \textit{for all} \quad 0 \leq i < n,$$

where $n \geq 0$. We refer to n as the length of the execution fragment ϱ. An infinite *execution fragment ϱ of TS is an infinite, alternating sequence of states and actions:*

$$\rho = s_0 \alpha_1 s_1 \alpha_2 \ldots \quad \textit{such that} \quad s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \quad \textit{for all} \quad 0 \leq i.$$

3.2.1 The Harrison, Ruzzo, Ullman Security Model

The *Harrison, Ruzzo, Ullman security model* (*HRU model*, [26]) is an operating system level computer security model which deals with the integrity of access rights in the system.

A protection system consists of the following parts: (i) a finite set of generic rights R, (ii) a finite set C of commands of the form:

```
command $\alpha(X_1,\ X_2,\ .\ .\ .,\ X_k)$
  if $r_1$ in $(X_{s_1}, X_{o_1})$ and
     $r_2$ in $(X_{s_2}, X_{o_2})$ and
        ...
     $r_m$ in $(X_{s_m}, X_{o_m})$
  then
     $op_1$
     $op_2$
        ...
     $op_n$
end
```

where α is a name, and $X_1, \ldots, k_m$ are parameters. Each op_i is one of the primitive operations

```
enter r into  $(X_s,\ X_o)$
delete r from $(X_s, X_o)$
create subject $X_s$
create object  $X_o$
destroy subject$X_s$
destroy object $X_o$
```

r_i are generic rights, and $s_0, \ldots, s_m$ and $o_0, \ldots, o_m$ are integers between 1 and k.

A *configuration* of a protection system is a triple (S, O, P), where S is the set of current subjects, O is the set of current objects, $S \subseteq O$, and P is an access matrix,[1] 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).

[1]An access matrix can be envisioned as a rectangular array of cells, with one row per subject and one column per object. The entry in a cell - that is, the entry for a particular subject-object pair - indicates the access mode that the subject is permitted to exercise on the object. Each column is equivalent to an access control list for the object; and each row is equivalent to an access profile for the subject.

While using the HRU model, the granularity of protection mechanisms, and the rules by which permissions can change are not of interest. On the other hand, I take inspiration from the HRU model in order to specifying and enforcing security policies that can be related to known vulnerabilities of web applications (i.e., extending the access control schema with information about the functionalities of a web application and the used information technologies). From this perspective the HRU model is well suited to be modified:

- The definition of the access matrix can remains unchanged.
- Commands can be instantiated with the functionalities offered by web applications.
- The set of generic rights has to be redefined in order to express the information that permit to relate web applications' functionalities to known vulnerabilities.
- The set primitive operations has to be changed accordingly to the changes performed on the other concepts.

Similar approaches to the HRU model (with the motivation about their non suitability to be applied in my approach) are:

- Biba model – The data integrity problem (i.e., if subjects can corrupt objects in a level ranked higher than them) is not easily modifiable since it deals only with read and write primitives.
- Bell-La Padula model – Data confidentiality and contents creation are not the focus of this thesis.
- Capability-based security – Sharing of capabilities (between user programs in operating system infrastructure) differs significantly from the web application environment.
- Clark-Wilson model – Preventing corruption of data items in a computing system is not relevant in this thesis, the data computation (and the associated policies) is not modeled.

3.2.2 Models of Web Applications

In this section, I introduce how the models of web applications are defined and the information that they contain.

The main goal of this thesis is to create a framework for model-based testing of web applications. When dealing with web applications, the experience of a penetration tester is sometimes stronger than the methodologies employed by security tools (Section 2.3). In this thesis, I aim to replicate

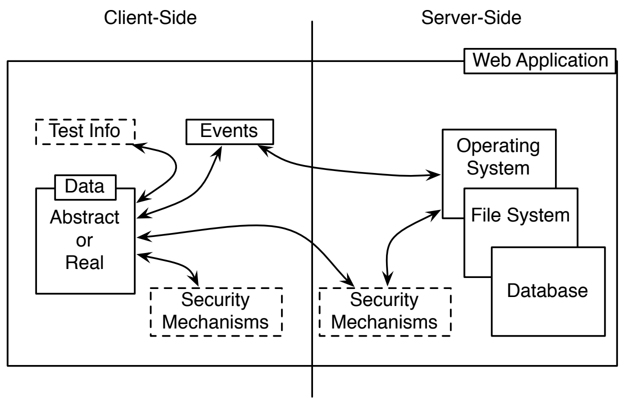


Figure 3.1: Features modeled regarding the data handled by the web application. Arrows depict the possible relation between data and the information about the storage and the management by the web application they refer to.

the experience of penetration testers in a model-based testing environment. Methodologies and approaches used by penetration testers will be the basis for the analysis of web applications and for the definition of models.

The analysis on how to define a model (② in Figure 1.1) starts from a completely different question: "How can I simplify the definition of the model for a penetration tester that never used model-based testing techniques?". In order to answer to this question, I started this thesis analyzing the data that penetration testers consider important during their tests:

- first of all, the knowledge and the data that users can handle while interacting with a web application are introduced (Section 3.3),
- then the behavior of the web applications (Section 3.4), and
- the security mechanisms and those information that can be used for security testing (Section 3.5).

The data gained from the analysis of these "macro-areas" are used in the models of web applications in order to define the set of atomic propositions AP of a transition system TS. As an anticipation of the data gained from this phase, in Figure 3.1 is depicted a high level representation of the features of a web application (on the client-side and the server-side) that are introduced in the model.

After the analysis of the different aspects of web applications to be modeled, in order to define a transition system (Section 3.2), the set of states S and the set of initial states I are defined in Section 3.6 adapting the HRU model (discussed in Section 3.2.1) to the case of web applications rather than operating systems.

A modification to the HRU model is also used to define the transition relation $\rightarrow$ and the set *Act*. Intuitively an action is a functionality of the web application that

(i) can be accessed through the user interface of the web application, (e.g., using a browser), and

(ii) changes the state of the client (this can be done by a direct call of the services on the server or via AJAX functionalities).

The informal definition of the actions is general enough to be applied to a large variety of features of a web application. In Section 3.7, I give an elucidation about the actions of interest in the framework.

Summing up, the model of a web application for security testing contains:

- A data structure containing the data that a web application (and its users) handles,
- the information about the storage and the management of these data by the web application,
- the functionalities that the web applications provides, and
- how these functionalities can be accessed by the users of the web application.

In the following, the components of the model are presented. With regard to the framework (Figure 1.1), the analysis of each component gives the means for the definition of the actions and can be used by security analysts in order to created their own models.

3.3 Users

3.3.1 Modeling a User

In the following, I describe what information about the users of web applications are modeled in the proposed approach. The desirable features that I want to describe for a user are: A user

- can interact with the web application (he has an unique identifier),
- can manage and use the data that the given web application uses,
- can interact with other users' data, and
- can interact with only one application at a time.[2]

[2]The framework presented here can be extended to users interacting at the same time with multiple applications, but this extension is not in the scope of this thesis.

Users: The set *UserName* is defined as the set of unique identifiers of users interacting with the web application. *UserName* contains the names of the users that have an account on the application. Nowadays the majority of the web applications allow anonymous browsing (i.e., the users are not logged in); in order to identify this type of users, I introduce a special label ($Anon$). An example of a possible instatiation of the set *UserName* is thus

$$UserName = \{Alice, Bob, Anon\}.$$

3.3.2 Users' Data

Abstract Data

One of the main problems with model checking techniques is how to choose the correct abstraction level of the model that has to be defined. Regarding the data that a user can handle, the security analyst has to start the analysis by establishing which abstract data are in the scope of the analysis (i.e., the data that are part of the analysis). These data are abstract representation of the data implemented in web applications that a user can handle. I define *MetaData* as a set of meta-data that the security analyst defines. The meta-data have to reflect the meaning of the data that the users use in order to interact with the web application. An example of *MetaData* is the following:

$$\begin{aligned} MetaData = \{&Credential, User\text{−}Id, Profile, Messages\text{−}List,\\ &User\text{−}Session, Uploaded\text{−}File, \ldots\} \end{aligned}$$

In this example, the intended meaning for each element of *MetaData* is:

- *Credential*: The identification data belonging to a user that prove his identity (e.g., username and password).
- *User−Id*: The identifier that the web application give to a user.
- *Profile*: The personal data associated with a user (e.g., a description of the characteristics of the user).
- *Messages−List*: A list of messages stored on the server that the user can access.
- *User−Session*: The abstract representation of the information interchange between the web application and the user.
- *Uploaded−File*: A file that can be uploaded on the server via the web application.

The introduction of the meta-data in the beginning of the modeling phase is meant to give to the security analyst the complete freedom in choosing the abstraction level best suited for her analysis.

Data Structure

The set *MetaData* has to be instantiated in an abstract record *UserData* containing the data handled by a user (i.e., the data associated with the elements of *MetaData*). The record *UserData* is thus the representation of the abstract information in *MetaData* (at some level of abstraction) that a user can handle. In other words, the record *UserData* is a container for the knowledge that a user can gain from the web application.

Definition 3 (*UserData*)**.** *Let* $|MetaData| = n$ *be the number of elements in MetaData, and*

$$BasicTypes = \{String, Int, Bool, \ldots\}$$

be the set of concrete data types and $StrucTypes = \{Profile, Credential, \ldots\}$ *the abstract types of the elements of MetaData. I define the record UserData as*

$$UserData = (field_1, ..., field_n)$$

where $field_i \in UserData$ *is an instantiation (at some level of abstraction) of the i-th element of MetaData and is in 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*

By definition, the syntax of the record is context free; in the next section, I will explain how to instantiate every variable for each user in *UserName*.

Example 3. As an example, a typical data structure and the corresponding types of its variables are:

```
UserData =
   (
   cred = (user, pwd),
   id,
   prof = (name, ...),
   mess = (m [, m]...[, m]),
   param,
   data = (d [, d]...[, d]),
   file
   )
```

Types:
cred : Credential
user : String
pwd : String
id : Id
prof : Profile
name : String
mess : List
m : Message
param : Parameter
data : List
d : Unknown
file : FileAddress

When a security analyst models a web application, some of these fields 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).

As an example of modification of the data structure, the list of messages can be modified in order to model a web application where each message has a title:

```
UserData =
    (
    ...
    mess = ((t,m) [, (t,m)]...[, (t,m)]),
    ...
    )
```

Types:
mess : List
t : Text
m : Text

The new list of messages can be used by the security analyst to (i) model a fine-grained data structure, and (ii) lower the abstraction gap between the model and the web application. △

Multi-Users Environment

Every user has access to his (and other users) meta-data through the user interface of the web application. In a multi-user environment, the security analyst has to be able to trace the "ownership" of the data and having only the set *UserData* is not enough for this purpose.

Definition 4 (*Data*)**.** *The set Data is defined by instantiating every element in UserData with each user in UserName, i.e.,:*

$$Data = \{x.y \quad | \quad x \in UserName \;\; and \;\; y \in UserData\}$$

The set *Data* thus contains all the possible data that users can handle during their interaction with the web application.

As stated before, these information will be used in order to model the knowledge of the users which must contain only constants. The possibility of modeling structured data result in the fact that the modeled users can know the subfields of these data without knowing the complete field (e.g., the name of a user can be known without the direct access to his profile) but the modeling freedom that the proposed framework allows the omission of some information from the record *UserData*. The security analyst can omit the information that she does not wants to be part of the analysis during the definition (in the model) of the set *UserData*.[3]

[3]The omission of data in the model can bring to a loss of expressiveness of the model itself, the experience of the security analyst plays a crucial role in selection what data have to be omitted.

In order to manage the knowledge of subfields, I considered the possible modeling choices in defining the subfields, i.e., how to manage the knowledge of the user (the variables in the model) programmatically. In the following, I present three possible solutions divided in two approaches.

Only one sub-field per field: In this case, I can delete the subfield and manage the information contained in the sub-field in a variable that contains the field and implicitly also the information contained in the subfield; for example if the user *Alice* has a profile with one subfield *name*, then I can delete *Alice.prof.Alice* and only use *Alice.prof*;

Many sub-fields for a field: In this case, I have to deal with the possible missing information in the subfields; I have two possibilities:

- *Data* contains only the known subfield(s) while the others are omitted. The missing fields are not part of the model, and they are not considered during the analysis.
- *Data* contains all the subfields, the ones that the user knows can be used and instantiated, the others are described through free variables (one for each subfield); until the values of these variables are not assigned they remain unknown and are not usable. In this scenario, the data reflect the real data of the web application but a computational load is given to the model checker since it has to process more variables.

As I will discuss in Chapters 4 and 5, the third choice is the best suited for my purposes since the first two create confusion in the modeling and concretization phases.

3.3.3 Users' Knowledge

In the previous section, I have explained how to model the data that the users can handle, in this section, I explain how the users can gain new knowledge and how different types of knowledge are modeled.

During the interaction with a web application, a user could have access to (or use) many types of knowledge. For example, he can use some information that he already knows, extract information from the web application itself, or, in borderline cases, even guess some information.[4] We describe these

[4]From the perspective of an attacker, allowing that some data are guessable means that the security analyst has to introduce in the model one (or more) ad-hoc actions that, after their use, permit this type of analysis. The only difference between the "gained" and "guessed" knowledge is the use of such actions in order to guess data, after that, the knowledge can be used independently from its type; differentiating the knowledge in gained and guessed has the purpose of introducing new possibilities in the analysis of web applications.

types of information with labels contained in the following set:

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

The intended meaning for the labels in K_{Source} refers the possible sources of knowledge that can be modeled:

- *Initial*: The user knows the data from the beginning of his execution (the origin of this knowledge is not discussed here and its definition is demanded to the security analyst).
- *Gained*: The knowledge about a data is gained through the interaction with the web application.
- *Guessed*: The value of the data has been guessed, this type of data will be used only by the attacker.

We can thus express the knowledge of the users as a set of triplets:

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

Each triplet in U_{Knows} states that user x knows the data d with some "assumption" k_{src} about the origin of the knowledge. In the following, no further analysis on the assumption on the knowledge origin is made.

3.3.4 Knowledge Evolution

In order to explain how the knowledge of a user can evolve, I will formalize the initial knowledge and, subsequently, describe two possible alternatives to define the knowledge evolution.

Initial Knowledge

When a user starts to interact with a web application, he already knows some information about the data that he will use. In the proposed framework every user in *UserName* has an initial knowledge that, as I have presented in the previous section, is labeled with *Initial*. Thus I can extract from the set U_{Knows} those data d such that $k_{src} = Initial$; the initial knowledge of a user $x \in UserName$ are those data

$$d \in Data \qquad \text{such that} \qquad (x, d, Initial) \in U_{Knows}$$

The data belonging to the initial knowledge are defined by the security analyst at the beginning of her analysis. In the following, I will use the set *initialK* as a container for the initial knowledge of a user.

Possible Evolutions

The evolution of a user's knowledge is possible through the assignment of labels to data in the set U_{Knows} (excluding the data already labeled as initial knowledge), this allow users to learn new information. The assignment of labels is made through *actions* (intuitively a functionality provided by the web application, for more details refer to Section 3.7); when an action is "performed" by a user the labels in U_{Knows} are changed accordingly to the action (Section 3.7).

The evolution of the knowledge depends on what the security analyst wants to model and how she decides to model it (refer to Section 3.3.4 for the modeling approaches about knowledge). As will become evident in the following sections, a security analyst can choose between different types of modeling approaches (Section 3.8) and in what to model about the different aspects of the web applications; these choices can result in different needs about the evolution of the knowledge.

As stated before, the knowledge evolves through the application of actions (that are also used for transitioning the transition system from a state to another). I can thus define what is the knowledge of the users at a given state:

Definition 5 (Knowledge snapshots). *Let $TS = (S, Act, \rightarrow, I, AP, \mathscr{L})$ be a transition system and $\varrho = s_0\alpha_1 s_1 \alpha_2 \ldots \alpha_n s_n$ (with $n \in \mathbb{N}$) an execution fragment:*

- *$U^{s_i}_{Knows}$ denotes the content of the set U_{Knows} at state s_i.*
- *The knowledge of the users evolves as*

$$U^{s_0}_{Knows} \xrightarrow{\alpha_1} U^{s_1}_{Knows} \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} U^{s_n}_{Knows}$$

  where $s_0 \in I$, and the set $U^{s_0}_{Knows}$ thus contains the initial knowledge.
- *The knowledge of a user "usr" at state s_i is the set:*

$$\{(x, d, k_{src}) \mid (usr, d, k_{src}) \in U^{s_i}_{Knows}\}$$

The two main options for the knowledge evolution are

- monotone knowledge or
- non-monotone knowledge.

Definition 6 (Monotone Knowledge). *Let $TS = (S, Act, \rightarrow, I, AP, \mathscr{L})$ be a transition system and ϱ an execution fragment where $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$, (with $n \in \mathbb{N}$). The knowledge of the users is monotone if*

$$U^{i}_{Knows} \subseteq U^{i+1}_{Knows} \quad \textit{for all} \quad 0 \leq i < n$$

The knowledge of the users is non-monotone otherwise.

With a monotone knowledge every piece of information that a user can learn cannot be forgotten; on the other hand, a non-monotone knowledge is well suited to be used in those cases where users can forget data during their execution of the web application. The *logout* functionality is a good example of these two possibilities:

- with a monotone knowledge, every information is permanently "stored", e.g., if the security analyst is modeling a session token, this will be accessible in all the interactions (part of a session or not) with the web application (even if it was not part of the initial knowledge);

- with a non-monotone knowledge, information can be deleted from the knowledge;[5] e.g., a session token can be deleted after a *logout* and cannot be used for future executions. For a matter of speaking, the "value" of the session token is forgotten in order to invalidate it. The same, can be applied to all session data in the case that the security analyst wants to model this particular behavior of web applications. In this setup, the initial knowledge can not be deleted; as part of the data that the security analyst defines in the model, I believe that the initial knowledge is the basis of the analysis and, thus, too important to be deleted.

A feature like the forgettable knowledge seems to disagree with many model checking techniques that "does not forget anything" during the executions. In my opinion, this feature has a crucial role in modeling web application, since in some cases some unrealistic assumption has to be made in order to model behaviors that require to forget.

Some examples of possible scenarios where non-monotone knowledge is required are:

- special functionalities that invalidate data after they are used (i.e., security checks on the data of the web application),

- gained information that are valid only for a session.

3.3.5 Attackers

In my approach, the concept of attacker is a bit blurry. Some research fields introduce in the analysis a powerful ad-hoc attacker (e.g., the Dolev-Yao attacker [19] for security protocols). In penetration testing, security analysts have to assume that (i) all the users of the system are potential attackers, and (ii) every point of the web applications can be used in order to enter dangerous data to or extract data. The main difference between this

[5]In the proposed framework, knowing a data implies that the data itself can be used during the interaction with the web application.

approach and other types of analysis is that an attacker is not introduced in the model as an entity or a channel.

As in penetration testing, the proposed framework uses one of the users of the web application as an attacker and, if needed, other users for the generation of attack traces with multiple users. Since every user can be an attacker, the analysis of web applications does not require the introduction of special users (or communication channels) than the ones that are used on the web application; the modeled users are thus the ones that interact with the modeled web application.

With these premises, some assumption about the interaction of the users with a web application can be made. In the following, I present two assumption that are used in the models of web applications.

Data can be dangerous When a user interacts with a web application, the developers are expecting that the interaction is made in an harmless way. From a penetration testing perspective, every interaction can be dangerous and must be treated as such. Following this idea, during the model checking phase of the framework, I have to make some assumption on the data that the users can handle:

- every time the web application permits to write a data, I assume that the same data can be modified in order to deliver a payload of an attack,
- when the web application takes in input a data, it is not sanitized, and
- when the web application displays a malicious payload, it is not sanitized.

In Section 4.2, these assumptions are used in order to define the goals that are used in the models.

Guessable data In Section 3.3.3, the possibility of guessing data has been introduced. When a security analyst introduces in the model the possibility of guessing data, she is defining how an attacker can differ his execution from the knowledge constraints that the web application has; in other words, she is giving to the attackers the means to attack the application. The introduction of such possibility for the attackers (i.e., the users of the web application) has not to be taken lightly, since it can give to the attackers too much freedom in their execution. As an example, if the knowledge of a data is used as a constraint for the access to certain functionalities, an attacker can guess these data (in the model) and this it is not possible on the real web application, then the model checking phase can return false positive counterexamples (i.e., attack traces that are counterexamples in the model but not possible to testable in the web application).

3.4 Web Applications' Behavior

3.4.1 Modeling Web Applications' Behavior

In the previous section, I have explained how to model users; in this section, the focus is on the web application (as the target of the tests) and its behavior. The desirable features that I want to describe for a given web application are: (i) it gives the means to access data (through an interface) to the users, and (ii) it relies on its back end machinery (i.e., server-side information technologies) for the storage and the management of the data.

In order to describe these features, my main focus is on the events of the web applications that

- a specific user causes them to happen (i.e., the events are triggered),
- through the use of the functionalities (i.e., actions) of the web application,
- regarding some data, and
- with respect to a specific "location" on the server.

3.4.2 Web Applications' Events

Events describe what it is happening to the web application's data (both on client-side and the server-side).

Web applications' events are related to actions; when an action α is performed some events take place (i.e., the user triggers some events through the use of the action). Let the syntax of the events be

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

where $x \in UserName$, *event* is the name of the event, $parameters \subseteq Data$, and *location* the location of the data regarding the web application information technology. An action could write some data on the web application's database, in this case the event $x.write(targetData, database)$ is related to that action.

In order to simplify the notation for the actions (Section 3.7) the events are specified in the target state of the action. As an example,

$$s_i \xrightarrow{\alpha_i + \{events\}} s_{i+1}$$

becomes

$$s_i \xrightarrow{\alpha_{i+1}} s'_{i+1}$$

where in s'_{i+1} for each event in $\{events\}$, a label expressing the *event* and its *location* is given to the event's *parameters*. As done before for the knowledge of the users, a set *Event* containing the possible labels for events is introduced.

An interesting instantiation of the set *Event* (that could be expanded by the security analyst) is the following:

$$\begin{aligned} \mathit{Event} = \{&\mathit{ShowDB}, \mathit{WriteDB}, \mathit{ShowFS}, \mathit{WriteFS}, \\ &\mathit{Exec}, \mathit{Edit}, \mathit{WriteSD}, \mathit{ShowSD}\} \end{aligned}$$

The intended meanings for the elements in *Event* when used to label a data 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 that read from a database).
- *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 the corresponding 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).
- *Edit*: The labeled data has been retrieved by the application for editing (e.g., a form that a user can edit).[6]
- *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 others pertaining a certain session.

The set *Event* gives us the means to define how the application works on the data.Through the set *Event*, the security analyst can model her understanding of how the data are managed by the information technology (IT) on which the web application relies on:

- databases that manage data,

[6]There is a slightly difference between a *ShowDB* (or the other types of "Shows") and an *Edit*; the first refers to the fact that the data are only displayed, the latter to the fact that a user can change the values of the data. We will use this event in those cases where the web application permits to modify the values of the data in a page and can display the same data (without the possibility to edit them) in another page (e.g., a form used in order to send the new fields of a profile).

- file systems for files,
- sessions for access control and volatile data,
- operating system that execute commands, and
- the web application's user interface that retrieves data from the users.

I can define the set of events that are related to the user and the data he is using:

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

Each triplet in WA_{Event} states that the event e happen on a data d and the user x triggered it.

As stated before, the events are triggered by the application of actions (that are also used for transitioning the transition system from a state to another). I can thus define what the triggered events at a given state:

Definition 7 (Events snapshots). *Let $TS = (S, Act, \rightarrow, I, AP, \mathscr{L})$ be a transition system and $\varrho = s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n$ (with $n \in \mathbb{N}$) an execution fragment:*

- *$WA^{s_i}_{Event}$ denotes the content of the set WA_{Event} at state s_i.*
- *The events triggered by a user "usr" at state s_i is the set:*

$$\{(x, d, e) \mid (usr, d, k_{src}) \in WA^{s_i}_{Event}\}$$

3.4.3 Events and Knowledge

Initial Events

When a user starts his interaction with a web application usually he begins with the home page, thus the security analyst can use the home page as a starting point for the tests (it is not unlikely that the starting point of the tests has to be different from the home page, in this case she can use another part of the web application as a starting point). In Section 3.3.4, I assumed that the security analyst can define the initial knowledge of the users, the same is possible for the events. The initial events have to be aligned with the decided starting point, and have to be changed in the case the security analyst decides to change the starting point of her analysis.

Knowledge Evolutions Through Events

When an event is triggered by a user, his knowledge (Section 3.3.3):

1. changes according to the event, and
2. has to be enforced to the data structure where the knowledge is "stored" (Section 3.3.2).

In other words, the events can refer to subfields of a data, in scenarios like this, some policies have to be introduced in order to decide the scope of the event (i.e., what data are affected by the event). I present two policies for the knowledge evolution:

- **Writes** When a *WriteDB* of a structured type occurs, then the write is also applied to the corresponding basic type(s) (if present). For example, if a profile is written on the database, also its subfields are written at the same time.
- **Shows** When *ShowDB* of a structured type occurs, then the agent learns the corresponding basic type(s) (if present). For example, if a profile is shown on the database, also its subfields are shown at the same time.

As stated in Section 3.3.4, the security analyst has two possibilities for the evolution of knowledge in the model:

- a monotone knowledge and
- a non-monotone knowledge.

While the first has to be used in those scenarios where users do not forget the data that they have access to, the latter is used when users can forget the data, e.g., between different sessions (in this case, the logout functionality plays a crucial role). As stated in Section 3.4.3, also events play a central roles in the possible evolution of the knowledge since users can learn new data from during the interaction with the web application. We define this feature with a fact that has to be implemented in the model.

Definition 8 (Knowledge Evolutions Through Events)**.** *Let $i, j \in \mathbb{N}$ with $j = i+1$, $s_i, s_j \in S$ two states of TS, i.e., the matrices M after the application of, respectively, i and j actions (see Section 3.7), $x \in UserName$ a user, $gainK_{s_i}$ the set defining the knowledge that is gained from the interaction with the web application for the user x in the state s_i (the same is valid for s_j), and $d \in Data$:*

$$gainK_{s_j} = gainK_{s_i} \cup d \in ShowDB \cup d \in ShowSD \cup d \in ShowFS$$

3.5 Security Mechanisms & Testing-Related Information

In the previous sections, I have explained how to model users' knowledge and web applications' behaviors. These information alone are too general to allow a complete analysis of the security of web applications (and afterward the actual tests). Penetration testers often target security mechanisms and technologies that are not expressed by the label introduced before, this requires an additional analysis of these aspects of penetration testing.

In this section, I introduce two notions: *security mechanisms* and *testing-related information*. The first (Section 3.5.1) refers to those 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 IT), the latter (Section 3.5.2) refers to those information that are interesting from a testing perspective but are not part of the knowledge or the behavior of the web application. With these two concepts, I will show how a security analyst can introduce in a model her knowledge about security measures (i.e., how web applications enforce a defense against attacks) and possible attacks (i.e., tests that are not drivable from the data or the events). As will become evident in the following sections, the introduction of these information in the model is equivalent to decorating the actions (i.e., the functionalities of the web application) of the transition system with labels.

3.5.1 Modeling Security Mechanisms

The modeling of a web application requires the integration of the possible security mechanisms that are enforced over the data that the application handles. Penetration testers usually perform ad-hoc tests for different functionalities, this "know how" is important for the success of a test and is not drivable from the data or the events that I have already introduced in the model, as an example, the users' privileges (i.e., their roles) or the session management for restricted ares are information not drivable from the data and the events already introduced in the model).

The OWASP testing guide [42] states that a generic security test suite might include security test cases to validate both positive and negative requirements for security controls such as:

- Authentication & Access Control
- Input Validation & Encoding
- Encryption
- User and Session Management
- Error and Exception Handling
- Auditing and Logging

In Section 3.5.3, I focus the analysis on those security mechanisms (enforced through/on some data) that refer to the above classes.

3.5.2 Modeling Testing-Related Information

Having introduced the security mechanisms into the model sometimes is not enough in order to have a good coverage of the penetration tests that a web application has to pass in order to have a good measure of its security. As an example, let's take into account the growing use of AJAX in nowadays web applications: This type of functionalities suffer from a variety of attacks (see Section 2.4.3 for further information) and penetration testers leverage these functionalities in order to attack web applications. Likewise, I aim to introduce in the model information like "use of AJAX functionalities" in order to improve the testing of web applications. The information that I am aiming to introduce in the model are those that came from the experience in penetration testing of the security analyst. These information can be used in order to (i) improve the attack surface (i.e., the sum of the different points where the attacker can try to enter data to or extract data from an environment), and thus the overall testing capability and efficacy of the framework, in the model and not only during the testing phase, and (ii) reuse the expertise of the security analyst in different models.

3.5.3 Security Mechanisms' Data

As stated in Section 3.5.1, security mechanisms are enforced over the data; this does not mean that they are only usable on the data data structure for those data managed by the web application and accessible by the users (Section 3.3.2).

In this section, an additional data structure is introduced. The new data structure is meant for those data that belong to the users but implement the security mechanisms of the web application.

As explained in Section 3.3.2, the security analyst can model the data on different abstraction levels. The same is also applicable to the security mechanisms that are implemented in web applications. In the following, I present what abstract security mechanisms are of interest in this thesis, the data structure to store these information, and later, how security mechanisms and testing-related information are introduced in the model.

Abstract Security Mechanisms

While modeling the security mechanism, the security analyst has to take into consideration a number of factors, for example:

- A web application can implement a multitude of security mechanisms.
- Not all the security mechanisms are implemented on data that the user can see.

- Some security mechanisms could be implemented as "actions" to be performed and not as data to be used.
- Sometimes only a subset of the security mechanisms are in the scope of the test.

As we will also see in the case studies (Chapter 5), I will use a security mechanism for "Authentication & Access Control" (along with *login* and *logout* functionalities) and one for "User and Session Management", respectively:

- an abstract data *session* that abstracts the idea of a session ID like JSESSIONID (JEE), PHPSESSID (PHP), and ASPSESSIONID (Microsoft ASP), and
- an abstract data *userType* that abstracts some data implemented in the web application in order to differentiate types of users interacting with it.

Security Mechanisms Data

In order to store the security mechanisms' data, the security analyst can add them to the web application data *UserData* (see Section 3.3.2). As an example, the following two are possible data that can be added to the set *UserData*

- *session* models a session ID that is created for a certain user, and
- *userType* defines the specific role of a user.

The more scrupulous readers should now expect a section about the "testing-related information" but this section is missing. Such information does not give rise to data that have to be inserted in the model but only, as I will explain in the following section, to assertions on the data already present in the model.

3.5.4 Assertions on Security Mechanisms & Testing-Related Information

Every time a user interacts with a web application, the interaction itself is made via a browser. Even if the user is not aware of what is happening, from a security perspective, a lot of information can be extracted from a web application about how its data are handled and how the security mechanisms are implemented. A security analyst can define assertions (in the form of labels) about these information in order to define the set *Assertion*.

The strategy that a security analyst can follow while 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 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 that I use in the case studies presented in Chapter 5:

$$\begin{aligned} \mathit{Assertion} = \{&\mathit{Granted}, \mathit{Checked}, \mathit{AJAX}, \mathit{Sanitized}, \\ &\mathit{Admin}, \mathit{User}, \mathit{Echoed}, \mathit{PageIncluded}\} \end{aligned}$$

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

- *Granted*: 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 before been saved or displayed.
- *isAdmin/isUser*: If the role of the users of the web application is checked (these assertions defines 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.[7]

[7]In Section 3.4.2, I have introduced the event *ShowFS* that could be seen as a repetition of *PageIncluded* described here. The two concept are indeed similar but they differ in the fact that I see the event *ShowFS* as some data that are parsed to be included in a page, while the assertion *PageIncluded* refer to the actual files where web pages are saved (e.g., files with extension html, php, or asp).

This type of information can be used by the security analyst in order to derive from the model of a web application those execution fragments ϱ that bring the system in a state where penetration testing has to be used (i.e., the actual attack is tested via penetration testing techniques), as an example:

- *Echoed* for XSS,
- *PageIncluded* for data harvest and alike, and
- *AJAX* for the ones presented in Section 2.4.3.

Through the labels contained in the set *Assertion*, the security can define how security mechanisms and testing-related information are related to the application's functionalities and the data that these functionalities use.

The labels in *Assertion*, that a security analyst define, refer to assertions that are related to actions and could have some *parameters* (subset of *UserData*). As done for the set *Event*, the labels are specified in the target state of the action where for each assertion, the label expressing the assertion is given to the assertion's *parameters*.

I can thus define the following set:

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

stating 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.

As stated before, the assertions are defined for the web application's actions (that are also used for transitioning the transition system from a state to another). I can thus define what the assertions at a given state:

Definition 9 (Assertions snapshots)**.** *Let $TS = (S, Act, \rightarrow, I, AP, \mathscr{L})$ be a transition system and $\varrho = s_0\alpha_1 s_1\alpha_2 \dots \alpha_n s_n$ (with $n \in \mathbb{N}$) an execution fragment:*

- *$SEC^{s_i}_{Assertion}$ denotes the content of the set $SEC_{Assertion}$ at state s_i.*
- *The events triggered by a user "usr" at state s_i is the set:*

$$\{(x, d, e) \mid (usr, d, k_{src}) \in SEC^{s_i}_{Assertion}\}$$

Knowledge Evolutions Through Assertions

In Section 3.5.4, I give an example of how a security analyst can define security mechanisms and test related information, i.e., describe how "security" is enforced over the data that the web applications use and how these data are managed.

Assertions do not play a direct role in knowledge evolution, since users do not gain direct knowledge from these information. As I will show in Section 3.7, from a security analyst perspective, assertions are mainly used in order to control which actions can be executed (i.e., what users are allow to do during their executions) and for the writing of goals (Section 4.2).

Initial Assertions

When a user starts his interaction with a web application some initial assumptions can be made about the initial state of the security mechanisms. One of the assumption that I have used in all the case studies is that either one of the users has a valid session with the web application (i.e., the web application granted him a session token) or the anonymous user starts the interaction with the web application. This came from the fact that my modeling choices wanted to differentiate the possible actions that can be performed inside a session, and to have some control about the user that starts the interaction with the web application.

Even if the grant of a session to a user is one of the features that a security analyst has to take care of, it is not impossible that a security analyst could choose to start the interaction from a point of the web application where the initial security assertions play a crucial role in the model checking phase. In this case, the security analyst has to define which are the initial assertions.

3.5.5 Atomic Propositions

I can now define the set of *atomic propositions* AP of the transition system TS.

Definition 10 (Atomic Propositions)**.** *I define the set of atomic proposition as:*

$$AP = \{X \mid X \in U_{Knows} \text{ or } X \in WA_{Event} \text{ or } X \in SEC_{Assertion}\}$$

where U_{Knows} *has been defined in Section 3.3.3,* WA_{Event} *in Section 3.4.2, and* $SEC_{Assertion}$ *in Section 3.5.4.*

3.6 States

In the previous sections, I have presented what type of information I want to model about users, web applications, and security mechanisms and testing-related information, in order to define the set of atomic propositions AP (Definition 10). In this section, I define the states of the transition system (S).

3.6.1 States of a Web Application

Every state $s_i \in S$ describes a particular snapshot of the web application regarding the information about:

- The users (in *UserName*) and their knowledge in $U^{s_i}_{Knows}$ (Definition 5).
- The triggered events (in *Event*) on the data ($WA^{s_i}_{Event}$, Definition 7).

- The assertions (in *Assertion*) about security mechanisms and testing-related information $SEC^{s_i}_{Assertion}$ (Definition 9).

A state describes what is happening client-side during the interaction of a user with the web application, with additional information about the server-side technologies used.

As presented in Section 3.2.1, the HRU model uses an access matrix P with a row for every subject, a column for every object, and right R that can be assigned to P's cells.

The elements of the sets U_{Knows}, $SEC_{Assertion}$ and WA_{Event} refer to information that relate users, data and labels (respectively K_{Source}, *Event* and *Assertion*). In order to trace how these information evolve, I define a matrix M as:

Definition 11 (States)**.** *For a given web application, a* state *of the TS is an instance of the matrix M such that the rows names take values in UserName, the column in every element of Data and the labels in K_{Source}, Event and Assertion are assigned to M's cells.*

In other words, I define an access matrix M that (i) remodels the HRU model (Section 3.2.1), and (ii) merges the information contained in the sets U_{Knows}, $SEC_{Assertion}$ and WA_{Event}, i.e.,

- the subjects of the HRU's protection system formalize the users of the web application,
- the objects formalize the data (in *Data*) that the web application can handle, and
- rights over objects formalize labels that express the information about the knowledge, the web application behavior, security mechanisms and testing related information.

Definition 12 (Initial States)**.** *Let $TS = (S, Act, \rightarrow, I, AP, \mathscr{L})$ be a transition system and M the matrix that defines the sates (Definition 11). The set I of* initial states *is defined by the security analyst with the instantiation of the matrix M.*

Example 4. Recalling that the set *Data* is instantiated for every user of the web application (Definition 4), let $UserName = \{Alice, Bob, Anon\}$ and $Data = \{Anon.session, A.session, A.profile, B.profile\}$. The matrix M that instantiates the state of TS is:

	Anon.session	*A.session*	*A.profile*	*B.profile*
Anon				
Alice				
Bob				

△

In the following I will use the following notation:

- M is the matrix containing the states of TS, and
- $M[i, j]$ a cell of the matrix (where the first subscript is the row number and the second is the column number)

Example 5. The following two instances of a matrix M are possible states of TS:

	Anon.session	*A.session*	*A.profile*	*B.profile*
Anon				
Alice		*Granted*	*ShowDB*	
Bob				

	Anon.session	*A.session*	*A.profile*	*B.profile*
Anon				
Alice		*Granted*	*ShowDB*, *Edit*	
Bob				

△

The matrices shown in latter example are small instances of states (i.e., for a small number of data); for a given web application the numbers of rows and columns can be larger.

In the following, I discuss how a security analyst can define the initial states I of the transition system TS.

3.6.2 Initial States

An initial state defines the initial knowledge of the users (Section 3.3.4), the initial events (Section 3.4.3) and the initial state of the security mechanisms and testing related information (Section 3.5.4). In order to define the possible initial states, the security analyst has to define the possible initial matrices M that instantiate these values with regard to the tests she wants to perform. I denote an initial state (or initial matrix) with M^0.

Since the initial states can vary considerably from web application to web application, in the following I give some examples of initial states along with some clarification about their meaning.

Example 6. The empty matrix (e.g., see Example 4) is a valid initial state. The evolution of TS can start from an empty matrix; this means that the transition relation of TS does not need to satisfy any condition in order to move the system to another state. △

Example 7. The matrix

	Anon.session	*A.session*	*A.ID*	*A.profile*
Anon	*Granted*			
Alice			*Initial*	
Bob				

formalizes an initial state where the anonymous user starts interacting with the web application, and two users are present (one whose has his *ID* as initial knowledge). △

Example 8. The matrix

	Anon.session	*A.session*	*A.ID*	*A.profile*
Anon				
Alice	*Granted*		*Initial*	*ShowDB*
Bob				

formalizes an initial state where to the user *Alice* a session has been granted by the web application, *Alice*'s *ID* is part of her initial knowledge and the web application is showing her profile (that is retrieved from the database). This example shows the case where all three types of triplet that compose AP (i.e., U_{Knows}, $SEC_{Assertion}$ and WA_{Event}) are used. △

State Space

Since the cells of the matrix M contain a subset of AP, I am stating that each cell can contain multiple values and the state of TS can change also in the case that a value from AP is appended to the values contained in a single cell, i.e., the set of states S contains all possible instances of the matrix M.

Definition 13 (State Space)**.** *Let UserName be the set of usernames, Data the set of data, AP the set of atomic propositions, $n, m \in \mathbb{N}$ such that $n = |UserName|$ and $m = |Data|$ are respectively the number of rows and columns of the matrix M. The set S of states is defined as the union of all the possible instances of the m-by-n matrix M:*

$$S = \{ \quad M \quad | \quad \forall i < m, \quad \forall j < n \quad . \quad M[i,j] \subseteq AP\}$$

It is possible to calculate the number of possible elements of S (i.e., $|S|$). Let $n, m, p \in \mathbb{N}$ such that $n = |UserName|$, $m = |Data|$, and $p = |AP|$, the number of possible states in S is the number of elements in the power set of AP times the number of elements in the power set (written $\mathcal{P}$) of the elements of the matrix M (i.e., the cells of M):

$$|S| = \mathcal{P}(AP) \cdot \mathcal{P}(m \cdot n) = 2^{n \cdot m} \cdot 2^{p} = 2^{(n \cdot m)+p}$$

As an example, let $n = 3$ be the number of users, $m = 4$ the number of the modeled data, and $p = 5$ the number of atomic propositions. The number of states in the transition system defined with this parameters is $|S| = 2^{(2 \cdot 4)+5} = 131072$

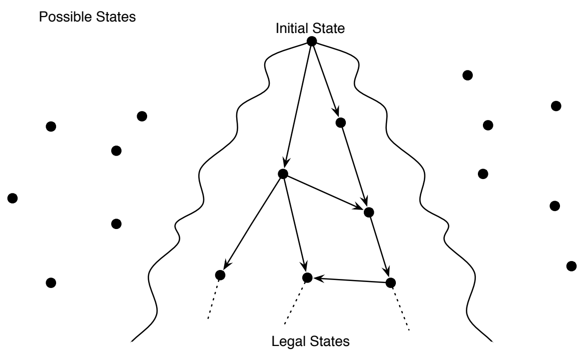


Figure 3.2: Representation of reachable states in a transition system.

3.6.3 Transitions and Reachable States

In the previous sections, I have described how the states of the transition system are modeled. In order to describe the reachable states of a transition system, the transition relation $\rightarrow$ (and the set *Act* of action names that describe the transitions) has to be introduced.

Transitions describe how the system evolves from one state into another. The security analyst defines the transitions through the definition of actions (presented in Section 3.7). The definition of the actions correspond to the definition of the transition relation ($\rightarrow$) and the set *Act* at the same time. As an initial definition, an action (i.e., transition) is a procedure that, given a state, checks if it is possible to apply the action (i.e., if the conditions to apply the action are satisfied), and calculates the next state of the transition system.

The *reachable states* of a transition system are those that are reachable through the application of all the possible transitions $\rightarrow$ from the initial states in I (i.e., initial states defined by the security analyst). In other words, the reachable states are the ones that are part of an execution fragment (Definition 2). In Figure 3.2, I depict the set of reachable states (immersed in the state's space) from an initial state through the application of actions.

3.7 Actions

In this section, I present how the security analyst has to define the transition relation and actions of the transition system in order to model the functionalities of the web application.

Web applications offer various types of functionalities, ranging from general purpose functionalities such as authentication, editing of private information or searching information, to specific functionalities such as reading

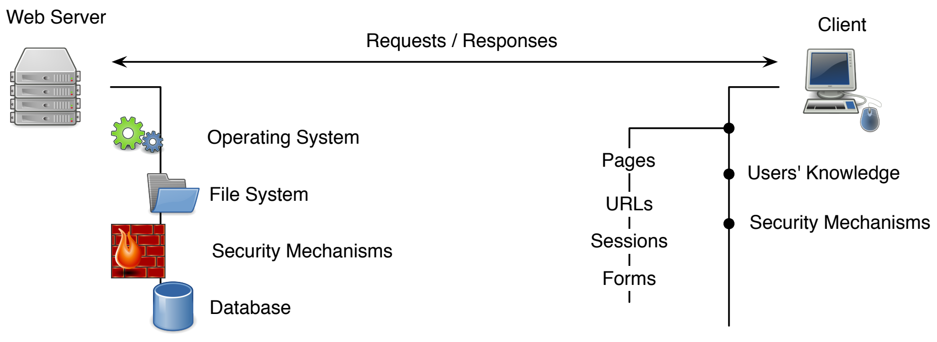


Figure 3.3: Features of interest in the modeling of web applications.

a newsfeed or purchasing goods from an online shop. During the modeling phase, I am not interested in all the implementation details of the single functionality in order to consume it; rather (as I have explained in the previous sections), I focus my attention on the data that the functionalities have access to, and the origin/use of these data on the web application.

As stated before, web applications' functionalities can vary in a multitude of ways, but some common features can be found (some of these have already been discussed in the previous sections) both on client-side and on server-side (as depicted in Figure 3.3). In the following, I present the features that I take into consideration during the definition of the actions from the web application perspective; the same is valid for the functionalities that the web application implements. The client-side of the web application permits the users to

- have access to the web pages in which the application is divided,
- interact with URLs (in the form of links or addresses),
- use some "hidden" data such as sessions and cookies,
- send/receive data through forms,
- have their own knowledge about the content and the data of the web application, and
- interact with the security mechanisms that the application implements;

while, the client-side of the application permits the users to

- access the web server where the application is stored,
- access the file system of the server, and
- use the database(s) that manages the data of the web application.

In Sections 3.3, 3.4 and 3.5, I have defined all the data that a model can contain. In the definition of the data, I have implicitly used a division into levels of abstraction of the data handled by web applications and their clients. In Figure 3.1, I depict the different features that I am modeling with regard to the data handled by the web application. With regard to the level of abstraction of the data:

- Data are always seen on the client side of the web application (even when I refer to server side technologies).
- Data can refer to information accessible and modifiable by the users (e.g., via a form); the security analyst can decide whether to abstract the data or keep the "real" ones (i.e., parameters used by the application that are not abstracted).
- Data can refer to information that are not accessible by the users (i.e., they are hidden to the users) and are used in the pages for security purposes. This set of data can be informally defined as the combination of the data that model the security mechanisms and the test related information.
- Data are "produced" by the users as part of their initial knowledge or by operations executed on the server side of the web application.

After this introduction, I discuss the functionalities that are of interest and how to model actions in the framework.

3.7.1 Functionalities of Interest

The functionalities of a web application in which I am interested are:

- Functionalities that change the state of the web application (and thus of the transition system) through the interaction with the information technologies that the web application uses (i.e., the database, the file system, etc.). These functionalities are common in web applications, since they are needed for the navigation through pages, showing or modifying information, and so forth. As an example, in a web forum the following functionalities are present: on the database a user can save texts or personal information (i.e., a profile), on the file system a user can save an image.
- Functionalities that are needed in order to model the security mechanisms implemented in the web application. These functionalities refer (in the majority of the cases) to those information that the users do not see (e.g., sessions and information stored in cookies) and are used in order to implement the information technology security. As before, let's take the "login" functionality as an example. Even if the login is

accessible through a form, the operations that concern a login phase are mainly security related, e.g. check if the username exists, if the password is correct, open a session for the user, and so on.

These two examples do not strictly divide the functionalities of a web application into categories, since most of them are implemented as a combination of the two.

Concluding, I will model actions that refer to functionalities that

- use server-side features of the web application, e.g.:
  - read/write from a database, file system,
  - execute commands,
  - change pages, etc.
- are used for some security reasons (and thus are of interest during a test).

3.7.2 Modeling Actions

Let the following set be a set of labels for the functionalities that I am modeling:

$$\begin{aligned} \mathit{functionName} = \{&\mathit{Login}, \mathit{ViewProfile}, \mathit{GetEdit}, \mathit{EditProfile},\\ &\mathit{ListId}, \mathit{Search}, \mathit{UpdateProfile}, \mathit{Logout}\} \end{aligned}$$

The elements in *functionName* refer to the functionalities implemented in the web application that are modeled as actions (in Figure 1.1 I labeled this step as "identification phase"). These actions have to be instantiated with respect to the data (contained in the set *Data*) of the web application. Even though the functionalities are common to multiple web applications, the data on which they are applied to can be different from case to case. In Table 3.1, I give an example of actions and parameters for a general user x.

Actions also describe the transition relation $\rightarrow$. The security analyst has to define it in the set *Action* (of actions) that is introduced in the model.

Definition 14. Action *An action* $\alpha \in \text{Action}$ *is defined as*

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

where

- $\mathit{name} \in \mathit{functionName}$*,*
- $\mathit{agent} \in \mathit{UserName}$*,*
- $\mathit{parameters} \subseteq \mathit{Data}$*,*

Table 3.1: An example of actions for a user x and some parameters.

High-level action	Functionality
$\mathit{Login}(x, x.\mathit{cred})$	A user $x \in \mathit{UserName}$ is authenticated on the web application using the credential $x.\mathit{Cred}$
$\mathit{ListId}(x)$	The known IDs of a user are listed by the graphical interface
$\mathit{ViewProfile}(x, \mathit{data})$	A profile is selected with respect to some data and displayed
$\mathit{GetEdit}(x, \mathit{data})$	An editable instance of DATA is retrieved
$\mathit{Search}(x, \mathit{value})$	A search engine is called with parameter value
$\mathit{Logout}(x)$	The user no longer needs access to the restricted area of the web application
$\mathit{UpdateProfile}(x, x.\mathit{prof})$	The action is defined for a particular functionality; in cases like this the action is instantiated for ad-hoc parameters

- *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.*

In order to write the actions in *Action* the following grammar has to be used:

```
name(agent,parameters)
   [if condition:]*
      tr
      [tr]*
end.
```

In the following, I present in detail how the sets *PrimitiveTransitions* and *Condition* are discuss.

3.7.3 Primitive Transitions

Intuitively, applying an action to the matrix M, correspond to the use of a functionality on the web application (being the HTTP protocol stateless this feature follows the possibility of sending requests to the web application without using the graphical interface).

The set *PrimitiveTransitions* contains the definition of the possible transitions of the transition system (i.e., the set of primitive operations that can be performed on the matrix M).

Each element in *PrimitiveTransitions* can be in two forms; let M be a matrix denoting a state of TS, $X \in AP$, $A \in UserName$ and $D \in Data$, the elements in *PrimitiveTransitions* are in the forms:

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

where the above is used when the transition is applied to a single cell of M, and the latter when the transition is applied to all the cells in the row A.

As an example, the following primitive transitions are used for the definition of actions in the case studies (Chapter 5):

- `Add` X `into` $M[A, D]$
  X is appended in the cell $M[A, D]$ (it could contain other values),
- `Del` X `from` $M[A, D]$
  X is deleted from the cell $M[A, D]$, and
- `Reset` M `for` A
  in the row A of M every value that differs from *Initial* or *Gained* or *Granted* is deleted.

In the last operation, the values of the knowledge are not deleted as an example of monotonic knowledge; in the case the security analyst wants to model a non-monotonic knowledge different primitive transitions have to be used; I also assume that the *Granted* assertion can be deleted only with the application of an action.

Each operation changes the values of the matrix M but in the case multiple operations are applied, as needed in the next sessions, I consider as a change of state only the result of the use of all the operations of an action, e.g., let M be a matrix denoting a state of TS, M_0, M_1, M_2, M_3, M' possible instances of M, and $op_1, op_2, op_3 \in PrimitiveTransitions$ operations of an action α:

$$M_0 \xrightarrow{\alpha} M' \;=\; M_0 \xrightarrow{op_1} M_1 \xrightarrow{op_2} M_2 \xrightarrow{op_3} M_3$$

where $M' = M_3$ as the result of the composite transition of all the operations in α.

Conditions

As stated above, the set *Conditions* contains conditions that have to be satisfied in order to perform the primitive operations associated with an action.

Let M be a matrix denoting a state of TS, $A \in UserName$, $D \in Data$ and $X \in AP$. A condition is an expression of the form:[8]

$$\texttt{if} \quad X \qquad \in / \notin \qquad M[A, D]$$

Since $X \in AP$, the definition of conditions expresses three possible scenarios

- $X \in K_{Source}$, i.e., conditions about the knowledge, if the action requires some specific knowledge to be performed; this type of conditions can restrict the evolution of the model regarding the knowledge of the users (i.e., *initial*, *gained*, and *guessed*);
- $X \in Event$, i.e., conditions about the events of the web application, if the action requires that an event occurred before it; this type of conditions can bound the evolution of the model regarding the behavior of the web application, e.g., the flow between the pages of the web application (if the security analyst wants to model this kind of behavior);
- $X \in Assertion$, i.e., conditions about the security mechanisms and test related information, if the action can be performed only in certain scenarios described through the security of the web application, e.g., a user with a session or with a particular role.

[8]In the following, the use of $=$ and $\neq$ will be used as an abuse of notation for $\in$ and $\notin$, even if the cells of the matrix can contain multiple values. We use this notation in order to stress the requirements of the actions.

The following example explains the definition of the action *Login*.

Example 9 (Defining the action *Login*)**.** It is straightforward that the action's *name* is "Login". We want to maintain the functionality general enough to be used by multiple agents, thus let x be the agent using it. We can choose to use as *parameters* both "username, password" and "credential" (instantiated for the user x). Assuming that we do not need to differentiate the parameters use "credential". We thus have:

$$Login(x, x.credential)$$

Usually, a login can be performed only if the user is not logged in yet (i.e., he is still anonymous to the application) and if he knows his credentials (i.e., the credentials are part of his knowledge):

```
if $M[Anon, Anon.session] = Granted$
if $M[x, x.cred] = Initial$
```

Once the above conditions have been fulfilled, I have to change the state of the transition system. First of all, I delete the previous events from the matrix (through the primitive reset), I then delete the *Granted* atomic proposition from the *Anon* user (this also means that I assume that a user can login only from an anonymous session) and I give the session to the user requesting it. Since the "credentials" are checked (the low-level mechanism is not important) I also add this information to the state and I can close the action:

```
Reset $M$ for $x$
Del $Granted$ into $M[Anon, Anon.session]$
Add $Granted$ into $M[x, x.session]$
Add $Checked$ into $M[x, x.cred]$
End
```

△

3.8 Modeling Approaches

In the previous sections, I have explained how I model the states and the actions of *TS*. As explained in Section 3.7.2, the conditions inside the action can be used by the security analyst in order to model

- the navigation between pages, and
- the data constraints of the application, i.e., what has to be sent for a specific functionality.

During the model checking phase, the matrix M is used in order to define the states of the web application regarding the functionalities that the web application implements. Since the functionalities can be accessed through HTTP requests, the information stored in the matrix M (i.e., the data that define a state of TS) can refer to “page” or “part of the UI” where some data are displayed or in which a user can access the functionalities of the web application.

In order to define the relations between “pages” and functionalities (i.e., how they coincide), I introduce the notion of views: how the actions correspond to the real implementation of the web application with regard to its pages. With the definition of the views, I want to give the means to the security analyst to model each functionality as a single action without the need of defining one action referring to multiple functionalities.

An example of views can be found in WebGoat (Section 5.1). The two functionalities *Login* and *ListId* can be defined; in the actual implementation, the two functionalities are used concatenated in order to show the page after a successful login, in other words, the security analyst has a concatenated view of the two actions.

The security analyst has two possibilities for defining the views of a web application:

- *Pre-definition*: the security analyst can merge the concatenated functionalities in the model defining a new action containing the final result or introduce a fact stating that the two actions have to be always concatenated.

- *Post-definition*: the definition of the concatenated actions remains the same as defining the single actions and the security analyst deals with the concatenation during the concretization phase.

Is intuitively correct that the analysis of a web application with both the approaches is correct since:

- the first restricts the research space and follows the flow between pages that the web application implements but is bounded by the constraints introduced with the concatenation of actions.[9]

- the latter can miss to catch the concatenation in the counterexamples (with the possibilities of having false positives counterexamples) but follows the stateless status of the HTTP protocol.

[9]Restricting the research space of the model checker in order to follow the flow between pages is correct for functional testing but it could bring to possible vulnerabilities to be discarded.

3.9 Small Conclusion

In this chapter, I have present a methodology for modeling web applications for security testing. A transition system suitable for the model-based testing of web applications has been defined and explained in its components:

- The data structure that the web application and its users use.
- The atomic propositions that permit to model knowledge, events and assertions.
- The states of the transition system as snapshots of the web application.
- The actions that permit the evolution of the transition system through the states.

In the following chapter, the focus is on how to use the transition system for model-based testing and on how fill the abstraction gap between the abstract tests and the actual implementation of web applications.

Chapter 4

Bridging the gap: From High-level to Low-level Verification

In the previous chapter, I have introduced how the transition system is defined for web application. In this chapter, I discuss how the transition system introduced in Chapter 3 is defined as an Alloy model (Section 4.1), in Section 4.2 the focus is on the definition of the security goals that the models have to satisfy. In Sections 4.3 and 4.4, I discuss the methodology and the implementation of the framework used to fill the abstraction gap between the abstract tests and the implementation of the web application. In Section 4.5, I discuss a vulnerability testing tool that can be used as test execution engine in the framework.

4.1 Defining the Models in Alloy

In the following section, I illustrate how models are defined using Alloy in order to be used during the model checking phase of the framework (③ in Figure 1.1).

4.1.1 Alloy

Alloy [1] is a structural modeling language based on first-order logic, for expressing complex structural constraints and behavior. The *Alloy analyzer* is a constraint solver that provides fully automatic simulation and checking. Alloy has been developed by the Software Design Group at MIT. The first Alloy prototype came out in 1997, and was a rather limited object modeling language. Later versions added quantifiers, higher arity relations, polymorphism, subtyping, and signatures (Alloy's structuring mechanism). The performance and scalability of the tool have gradually increased.

4.1.2 Models in Alloy

An Alloy model is a collection of constraints that describes (implicitly) a set of structures, for example: all the possible security configurations of a web application, or all the possible topologies of a switching network. The Alloy analyzer is a solver that takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, and to check properties of the model by generating counterexamples. Structures are displayed graphically, and their appearance can be customized for the domain at hand.

The statements that compose an Alloy model (and that I will use during the definition of web applications models) are:

- *Signatures* that define the vocabulary of a model by creating new sets;
- *Facts* that are constraints that are assumed to always hold;
- *Predicates* that are parameterized constraints, and can be used to represent operations;
- *Assertions* that are assumptions about the model that can be checked using the Alloy analyzer.

In the following, I use the above statements in order to define the models that the framework uses (② in Figure 1.1).

4.1.3 Data Used in the Model

In Section 3.3.2, I have defined the data that I want to introduce in the models of web applications:

- In Definition 3, the set *UserData*, of data handled by a user.
- In Definition 4, the set *Data* containing all the data that the users (in a multi-user environment) can handle during their interaction with the web application.

All the information contained in these sets compose the vocabulary of the Alloy model. I start by defining the set *UserData* with an abstract signature.

Example 10 (Abstract signatures)**.** As an example, the following code defines the data structure used in Example 3:

```
abstract sig Data {}
abstract sig Id, Credential, Name, Addr, UserType,
             Session extends Data {}
```

With the abstract signatures `abstract sig Data{}` I declare the set `Data` that contains no elements other than the ones in its subsets (if any). In the second line, I declare that the sets

```
Id, Credential, Name, Addr, UserType, Session
```

are disjoint subsets of `Data` (i.e., each element `extends` the set `Data`). As an example, `Id` ⊂ `Data`, `Credential` ⊂ `Data`, and `Id` ∩ `Credential` = ∅. △

Having introduced in the model the set *UserData*, I have to instantiate each variable for every user of the web application that I want to model. In the following example, I introduce the signatures for Alice, Bob and the anonymous user.

Example 11 (Users' signatures)**.** The signatures for Alice, Bob and the anonymous user are

```
one sig AliceId, BobId, NoId extends Id {}
one sig AliceCredential, BobCredential,
        NoCredential extends Credential {}
one sig AliceName, BobName, NoName extends Name {}
one sig AliceAddr, BobAddr, NoAddr extends Addr {}
one sig AliceSession, BobSession,
        AnonSession extends Session {}
```

Where the sets defined in Example 10 are partitioned. With the statement `one sig` I declare that the sets that I am defining are singleton sets. As an example, `Id` = { {`AliceId`}, {`BobId`}, {`NoId`} }. △

Since the field *Profile* is composed by the subfields *Name* and *Address* (*Addr* for short), I have to define first an abstract signature for the profile and then to instantiate it for each user.

Example 12 (Users' signatures II)**.** The abstract definition of the signature *profile*, and the instatiation for each user are:

```
abstract sig Profile extends Data {
  name: one Name,
  address: one Addr   }

one sig BobProfile extends Profile{} {
  name = BobName
  address = BobAddr   }

one sig JerryProfile extends Profile{} {
  name = JerryName
  address = JerryAddr }

one sig NoProfile extends Profile{} {
  name = NoName
  address = NoAddr    }
```

△

With all the data introduced in the Alloy model as signatures, I define the abstract data structure for the users as:

```
abstract sig User{
  profile: one Profile,
  id: one Id,
  name: one Name,
  credential: one Credential,
  session: one Session,
  initialK: set Data,
  gainK: set Data
}
```

and instantiate each user with his data, e.g.:

```
one sig Bob extends User{
    profile = BobProfile
    ID = BobId
    name = BobName
    credential = BobCredential
    session = BobSession
    initialK = credential + BobId +
                AliceId + AliceName + BobName
    gainK = NoData
}
```

These signatures correspond to the elements of *UserData*, instantiated in the Alloy model for each user, and to the two sets containing the knowledge of the user:

- *initialK*, containing the initial knowledge of the user (i.e., those information that are known by the user before interacting with the web application),
- *gainK*, containing the knowledge that a user gain during the interaction with the web application.

As I have introduced in Section 3.3.4, the users knowledge can evolve through events (Section 3.4.3). In order to introduce this feature in the Alloy model, I have to introduce the following fact:

```
fact {
 all s: State, s': s.next{
  s'.gainK[s.user] = (s.gainK[s.user] +
                      s.showDB +
                      s.showSD +
                      s.showFS ) &&
  (all u : User | (u != s.user)
   implies s'.gainK[u] = s.gainK[u])
 }
}
```

that defines how the set *gainK* evolves between the different states (introduced in the next section) of the model. As will be introduced in the next section, the set `gainK` is instantiated in each states for each user, i.e., `gainK[u]` ∀ `u` ∊ `User`. The fact specifies that for all states `s` and their successors `s'` (such that `s' : s.next`):

- For the user that is performing the action (i.e., `s.user`) his set `gainK` in the successor state contains
  - the knowledge already gained (i.e., `s.gainK[s.user]`) plus
  - the elements of `Data` that are displayed by the web application (i.e., the elements of the sets `s.showDB`, `s.showSD` and `s.showFS`).
- For the users that are not performing the action (i.e., `u != s.user`)
  - their sets `gainK` remain unchanged.

4.1.4 States

As introduced in Section 3.6 and more specifically in Definition 11, a state of the transition system is defined by a matrix M such that the row names take values in *UserName*, the column names in every element of *Data* and the resulting cells take values in AP. Since these sets can vary significantly from one web application to another, in order to simplify the modeling phase, I define the Alloy models' states and the initial state as reported in Table 4.1.

The definition of a state in the Alloy model (left handed part of Table 4.1) follows the following assumptions:

- In every state there is one user that had performed an action is specified in order to simplify the definition of goals (`user: one User`).
- The state reports the action that has been used to reach it (i.e., `action: one Action`).
- Each elements of AP is defined as a set that takes values in `Data` (e.g., `granted: set Data`).
- The set `gainK` is defined as a relation between the elements of `User` and the ones of `Data` (i.e., `gainK: User -> set Data`).

Since the set of atomic propositions (AP) is more likely to remain stable (i.e., change slightly between the tests made in a long period of time), the choice of modeling the states in such way simplifies the modeling activity of the security analyst, since the states' definition remains unchanged during the testing of different web applications.

In Table 4.1, I also show an example of initial state where `first.X` refer to the first state from which Alloy starts its execution:

Table 4.1: States definition and initial state definition for the Alloy models.

```
sig State {
  //User
  user: one User,
  action: one Action,
  //Sec. & Test Info
  grant: set Data,
  checked: set Data,
  AJAX: set Data,
  PageIncluded: set Data,
  echo: set Data,
  exec: set Data,
  //Web applications' Events
  showDB: set Data,
  writeDB: set Data,
  edit: set Data,
  writeSD: set Data,
  showSD: set Data,
  writeFS: set Data,
  showFS: set Data,
  noAttack: set Data,
  gainK: User -> set Data
}
```

```
fact{
  //User
  first.user = Anon
  first.action = NoAction
  //Controls' Predicates
  first.granted = AnonSession
  first.checked = NoData
  first.PageIncluded = NoData
  first.echo = NoData
  first.exec = NoData
  //Web applications' Events
  first.showDB = NoData
  first.writeDB = NoData
  first.edit = NoData
  first.writeSD = NoData
  first.showSD = NoData
  first.writeFS = NoData
  first.showFS = NoData
  first.AJAX = NoData
  first.noAttack = NoData
  (all u:User | first.gainK[u]
  = NoData)
}
```

- The first user is set with `first.user = Anon`.
- Since no action as been used, `first.action` contains the signature `NoAction` that is used only as a starting point for the model checker.
- Every element in AP is instantiated with the initial elements (e.g., `first.granted = AnonSession`) as discussed for the initial knowledge (Section 3.3.4), the initial events (Section 3.4.3), and the assumptions about the initial state of the security mechanisms and testing-related information (Section 3.5.4).
- The gained knowledge for the users is empty

```
(all u:User | first.gainK[u] = NoData).
```

In order to make possible the analysis of the actions for the Alloy analyzer, I have to introduce a fact stating which actions are usable.

Example 13. The following fact is introduce in order to define what actions are usable during the Alloy's executions.

```
fact {
  all s: State, s': s.next{
    Login[s,s'] or ListId[s,s'] or GetEdit[s,s'] or
    ViewProfile[s,s'] or UpdateProfile[s,s'] or
    Search[s,s'] or GetSearch[s,s'] or Logout[s,s']
  }
}
```

The fact states that for all states of the model, in order to switch from a state `s` to its successor (`s'`) an action (from all the possible actions defined in the model) has to be used (always respecting the conditions to be satisfied in order to use the single actions). As I will discuss in the following section, actions are defined as predicates (i.e., as functions that return a boolean value). With the definition of this fact I am stating that the conditions inside the actions have to be satisfied (both in `s` and `s'`) and thus I permit the evolution of the model from a state to another. △

4.1.5 Actions

As introduced in Section 3.7, an action is a functionality of the web application. In order to make the actions accessible to the Alloy analyzer, I declare their abstract signature:

```
abstract sig Action
```

and extend it with the names of the functionalities (*functionName* in Section 3.7.2) implemented on the web application. As an example, I can introduce in the model the following:

```
one sig Login, Logout, ListId, ViewProfile,
        GetEdit, UpdateProfile, GetSearch,
        Search, NoAction extends Action
```

In Section 3.7.2, I also have introduced the grammar used to write actions:

```
name(agent,parameters)
   [if condition:]*
      tr
      [tr]*
end.
```

and in Section 3.7.3 the primitives used as operations, i.e., "`Add` X `into` $M[A, D]$", "`Del` X `from` $M[A, D]$", and "`Reset` M `for` A".

In order to introduce the actions (defined in Section 3.7) in the Alloy model, their definitions have to be translated as Alloy predicates according to the following rules:

- `name` - unchanged.

- `agent` - as a condition on the session granted in `s.granted`, as an example,

  ```
  s.granted = AnonSession
  ```

  states that the action can be performed only if in the state `s` the session has been granted to the user `Anon`.

- `parameters` - as a condition on the knowledge of the user at state `s`, as an example,

  ```
  u.credential in u.initialK
  ```

  states that a user `u` has to have his credentials (i.e., `u.credential`) in his initial knowledge (i.e., `u.initialK`) for the action to be performed.

- `if condition` - as a condition on the knowledge of the user at state s.

- `Add X into M[A,D]` - as an assignment of value in *Data* (`M[A,D]`) to a variable in *AP* (`X`), as an example,

  ```
  s'.showDB = u.info
  ```

  assign to `s'.showDB` the value `u.info` (where `u` is a user and `info` is an abstract signature for a user data).

- `Del X from M[A,D]` - as the assignment of the value `NoData` to target element in *AP* (`X`):

  ```
  s'.showDB = NoData
  ```

- `Reset M for A` - as the assignment of the value `NoData` to all the elements of *AP* (that are not part of other operations) in the matrix.

In Example 9, the action login has been defined as

```
$Login(x,\ x.cred)$
  if $M[Anon, Anon.session] = Granted$
  if $M[x, x.cred] = Initial$
    Reset $M$ for $x$
    Del $Granted$ into $M[Anon, Anon.session]$
    Add $Granted$ into $M[x, x.session]$
    Add $Checked$ into $M[x, x.cred]$
End
```

The translation of this action in an Alloy predicate is:

Action Definition	Alloy definition
Login$(x,\ x.cred)$	`pred Login[s, s' : State]`
`if` $M[An, An.sess]$ = *Granted*	`s.granted = AnonSession`
`if` $M[x, x.cred]$ = *Initial*	`u.credential in u.initialK`
`Del` *Granted* `into` $M[An, An.sess]$	`s'.granted = NoData`
`Add` *Granted* `into` $M[x, x.sess]$	`s'.granted = u.session`
`Add` *Checked* `into` $M[x, x.cred]$	`s'.checked = u.credential`
`Reset` M `for` x	Assignment of `NoData` to all remaining elements

The complete action defined in Alloy is reported in Table 4.2.

In Section 3.8, I have introduced the possibility of having concatenated actions (i.e., the security analyst models every functionality of a web application but knows that two functionalities are always executed together). To make this possible, in the case studies in Chapter 5, I have introduced an Alloy fact stating which actions have to be concatenated. As an example, the following fact is used in order to force the application of the action `ListId` after the action `Login`:

```
fact {
  all s: State, s': s.next{
    Login in s.action implies ListId in s'.action
  }
}
```

Introducing this fact means that every evolution of the transition system not compliant with it is discarded by the Alloy analyzer.

Introducing Alloy facts as the one presented above means to introduce a constraint on the possible selection of the actions during the execution of the Alloy analyzer. Since the actions in the model are predicates, the security analyst has to be sure that in the concatenated action the conditions, that needs to satisfied in order to execute the action, are met. If some contradiction is present in the constrains, the model checker will return those counterexamples that are compliant with the contradiction. As an example, if the conditions in a `ListId` are not satisfiable after a `Login`, then the model checker will be unable to perform a `Login`, since, after its use, it is impossible to use the `ListId` (as the fact requires).

Table 4.2: Example of definition of an action in the Alloy models.

```
pred Login[s, s' : State]{
  one u : User-Anon |
  s'.action = Login &&
  s.granted = AnonSession &&
  u.credential in u.initialK &&
  s'.user = u &&
  //SecMec TestInfo
  s'.granted = u.session &&
  s'.checked = u.credential &&
  s'.pageInclusions = NoData &&
  s'.echo = NoData &&
  s'.exec = NoData &&
  s'.noAttack = NoData &&
  s'.AJAX = NoData &&
  s'.Sanitised = NoData &&
  // Events
  s'.showDB = NoData &&
  s'.writeDB = NoData &&
  s'.edit = NoData &&
  s'.writeSD = NoData &&
  s'.showSD = NoData &&
  s'.writeFS = NoData &&
  s'.showFS = NoData

}
```

4.2 Specifying Security Goals

In the previous section, I have introduced all the concepts for the definition of the Alloy models. In this section, I introduce the last concept needed in order to be able to do a security analysis of the web application: Security goals (in the following "goals").

From a security analyst perspective, the definition of a goal is equivalent to defining 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 (i.e., the execution fragments of *TS*)

starting from an initial state. In other words, a goal models the flaws that the security analyst is testing. Goals are written over the set of atomic proposition AP and can represent:

- access control rules,
- application logic flaws, and
- known vulnerabilities.

Definition 15 (Goal: General definition). *Recalling that a state in the transition system TS is an assignment of atomic proposition AP in the matrix M, and M^i is the state after the application of i actions, a goal:*

- *has a unique name (identifying the vulnerability to be tested),*
- *gives rise to a logical formula that tests some condition on the state M^i (for more complex goals, it is possible to use multiple indexes, e.g., M^i and M^j), and*
- *is used by the model checker in order to return an attack trace (i.e., an execution fragment ϱ).*[1]

In order to define goals, I list in Table 4.3 some operators (and the associated connectives) that can help the security analyst. Most common modeling languages are capable of dealing with these operators (along with the general definition of a goal), which can thus be used with different model checkers

In the proposed framework, the security analysts have the possibility of specifying different types of goals; in the following I give some examples and I discuss various typologies of goals. In order to write the goals, I make some assumptions (from an attacker perspective) about the atomic propositions used:

- A *WriteDB* (along with the other "writes") of a data d is rewritten as the malicious write of a modified data d (the name of the data remains the same) containing the payload of the attack.
- The input data d of a *WriteDB* (along with the other "writes") is not sanitized by the web application.
- A *ShowDB* (along with the other "shows") that displays a malicious payload does not sanitize the output (if not stated through the testing related information *Sanitised*).

[1]Stating that the model checker returns an execution fragment means that I always deal with a trace starting from the initial state, ending in the state where the security property is violated and (if the case) containing the states where part of the security property is violated (i.e., in those cases where the security property has to be valid for the traces of *TS*)

Table 4.3: Operators used for specifying goals.

Operator	Connective	Explanation
$\neg$	$!f$	negation
$=$	$f_1 = f_2$	equality
$\neq$	$f_1! = f_2$	inequality
$\wedge$	$f_1 \& f_2$	conjunction
$\vee$	$f_1 \| f_2$	disjunction
$\Longrightarrow$	$f_1 => f_2$	implication
$\forall$	forall $x_1 \dots x_n.f$	universal quantification
$\exists$	exists $x_1 \dots x_n.f$	existential quantification
neXt	$X(f)$	in the next state
Yesterday	$Y(f)$	in the previous state
Finally	$<> (f)$	at some time in the future
Once	$< - > (f)$	at some time in the past
Globally	$[](f)$	always
Historically	$[-](f)$	at all times in the past
$\in$	x_1 in X	set inclusion

4.2.1 Access Control Goals

As I have discussed in Section 2.4.1, access controls enforce decisions about whether a request to access a content or a function from a specific user has to be granted or not. In the following, I explain how to define the security goals of the vulnerabilities and attacks discussed in Section 2.4.1. Since the definition of goals depends on the security analyst, different (or equivalent) goals can be defined.

Vertical privilege escalation

As introduced in Section 2.4.1, a *vertical privilege escalation* occurs when a user can perform functions that his assigned role does not permit him to.

In order to test these vulnerabilities, I assume that the security analyst takes care of labeling the actions of the transition system with labels that reflect the typology of users that can perform them. With the label of the action differentiating the users, I can test whether a vertical privilege escalation is possible using a user with less permissions and making him perform the actions that require higher permissions.

Example 14 (Vertical privilege escalation)**.** As an example, let *administrator, manager and employee* be the labels used in order to differentiate the roles in access control scheme enforced on the application, and the possible action for each category be respectively:

1. *manageServer, addUser, deleteUser*

2. *createTask, deleteTask, completeTask, deleteMessage*

3. *addMessage*

In order to test a vertical privilege escalation, the tester should try to perform the sets of actions 1 and 2 with a user with privileges as "*employee*", and the set 1 with a "*manager*". △

Definition 16 (Vertical privilege escalation)**.** *Let $d \in Data$ be a data of the web application, $x \in UserName$, $Label$ a set of labels defined by the security analyst for the model, $l \in Label$, the matrix M^0 the initial state, and M^i the matrix after the use of i actions. A vertical privilege escalation occurs when there is a state M^i at the end of an execution fragment*

$$\varrho = M^0 \alpha_1 M^1 \alpha_2 \dots \alpha_i M^i$$

such that l is associated with α_i and the user performing α_i has not the needed permission.

Instantiated goal: Since the goal defined before is general, it has to be instantiated regarding the model used during the analysis. As an example, the following goal has be used in order to test the model presented in Section 5.1.2:

```
assert AdminAction {
  no s : State | some d : Data-NoData | some x : User{
  s.granted = x.session &&
  d in s.writeDB &&
  s.guessedK[x] = NoData &&
  isAdmin in s.checked
  }
}
```

The goal is stating that I want an execution fragment where the fact of being admin is checked. From a model checking perspective, the fact of having a trace where an admin functionality is used is not a real flaw of the web application, on the other hand, having this type of functionalities available becomes interesting during the testing phase. Introducing this simple goal reflects the choice of maintaining the model simple (i.e., omitting the needed checks in order to handle the users' roles) while shifting the resolution of the problem to the concretization phase (where I have the final result about the usability of these functions by a user with less privileges).

Horizontal privilege escalation

As introduced in Section 2.4.1, a *horizontal privilege escalation* occurs when a user can view or modify resources to which he is not entitled. As a comparison with the vertical privilege escalation, in this case the resource can be accessed by other users with the same role but not by the user we are testing. Since this type of vulnerabilities can vary significantly between different web applications, it is possible to express the related goals in different ways. In the following, I give examples of some goals in this category.

Secrecy of a data When considering the horizontal privilege escalation flaws, one of the issues that the model checking can help to tackle is the secrecy of data.

Example 15 (Horizontal privilege escalation)**.** Let's assume that we are modeling a web application where users have profiles containing private information such as credit card numbers and addresses. These data can be used during the interaction with the web application but cannot be seen (i.e., accessed) by a user different from the rightful "owner". If this circumstance can happen during the interaction with the web application, the secrecy of the private data can not be maintained and thus a horizontal privilege escalation has occurred. △

Definition 17 (Horizontal privilege escalation)**.** *Let $x, y \in UserName$, $x \neq y$, $PrivateData \subset Data$, $d \in PrivateData$ be a data of the web application that must remain private, the matrix M^0 the initial state, and M^i the matrix after the use of i actions. A horizontal privilege escalation occurs on the web application when there is a state M^i at the end of an execution fragment $\varrho = M^0 \alpha_1 M^1 \alpha_2 \ldots \alpha_i M^i$ such that $ShowDB \in M^i[x, y.d]$.*

In other words, I am expressing this goal as a secrecy assertion on a data:

```
assert Secrecy {
  no s : State | some d : Data-NoData | some x,y : User{
  s.granted = x.session &&
  x != y &&
  y.d in s.showDB
  }
}
```

The goal is general, i.e., it checks if all the data of the web application cannot be seen by a user different from the owner of the data. In order to focus the tests, a security analyst can target the data that has to remain private, i.e., `some d : PrivateData` where `PrivateData` is the signature containing the data that has to remain private.

Guessed data As explained in Section 2.4.1 ("Insecure ID's"), a horizontal privilege escalation can occur when an attacker can guess the IDs used by the web application. The security analyst can test this type of flaws during the concretization phase (see Section 4.3), but in order to do that, she has to introduce in the model some ad-hoc actions in order to make the users capable of guessing data.

Let's assume that the user can see some *IDs* and use them to access another user profile. In this case I can search for an execution fragment that will be instantiated with a list of *IDs* (in a similar way as a brute force attack) searching for a matching profile.

Definition 18 (Horizontal privilege escalation: Guessed data). *Let $x, y \in UserName$, $x \neq y$, $d_1, d_2 \in Data$ be data of the web application, the matrix M^0 the initial state, and M^i the matrix after the use of i actions. A horizontal privilege escalation occurs on the web application when there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1 M^1 \alpha_2 \dots \alpha_i M^i$ such that:*

$$Guessed \in M^i[x, x.d_1] \quad \wedge \quad ShowDB \in M^i[x, y.d_2]$$

The goal states that a data d_1 has been guessed by the attacker in order to access the data d_2. As an example, an instantiated goal is:

```
assert GuessedId {
  no s : State | some d : Data-NoData | some x : User{
  s.granted = x.session &&
  d in s.guessedK[x] &&
  d.(~id).profile in s.showDB
  }
}
```

The goal has been instantiated for a web application where the security analyst has decided to check if guesses are possible for a user that

- has a valid session on the web application (`s.granted = x.session`),
- has some ad-hoc actions to guess data (`d in s.guessedK[x]`), and
- these data are used in order to select other users' profile (`d.(~id).profile in s.showDB`).

Forced Browsing Past Access Control Checks

This type of attacks aims to enumerate and access resources that are not referenced by the web application. I choose to define a goal that can be used in order to test the web application for different types of attacks, i.e., with the introduction in the model of the information "*PageIncluded*" (i.e., in the URL/page there is a direct reference to a file that is rendered by the browser and displayed as a page) I can search for a valid entry point for those attacks that use files or their addresses. Among these attacks it is worth to mention the forced browsing (Section 2.4.1).

Definition 19 (Forced Browsing Attack)**.** *Let $x \in$ UserName, FileAddress $\subseteq$ Data, $d \in$ FileAddress be a file used by the web application, the matrix M^0 the initial state, and M^i the matrix after the use of i actions. A forced browsing attack can be launched if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1 M^1\alpha_2 \dots \alpha_i M^i$ such that:*

$$\mathit{PageIncluded} \in M^i[x, x.d]$$

The goal, in Alloy formalism, is translated as:

```
assert FileInclusion {
  no s : State | some d : FileAddress | some x : User{
    s.granted = x.session &&
    d in s.pageInclusions
  }
}
```

Since the methodologies used to test path traversal attacks (Section 2.4.1) are similar to the ones for force browsing attacks, the goal introduced in Definition 19 will be used also for path traversal attacks. The differences in the payloads used by the two attacks are dealt with during the concretization phase.

File Permissions

As introduced in Section 2.4.1, in order to find “file permission” flaws, the attacker has to gain access to files that are not intended to be presented to web users. This attacks can occur when an attacker can specify a path used in an operation that uses the filesystem. By specifying the resource, the attacker gains a capability that would not otherwise be permitted and enables him to access or modify protected system resources.

Example 16 (Path Manipulation from [49])**.** The following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker could provide a file name such as `../../tomcat/conf/server.xml`, which causes the application to delete one of its own configuration files:

```
String rName = request.getParameter("reportName");
File rFile = new File("/usr/local/apfr/reports/" + rName);
...
rFile.delete();
```

△

Since the security analyst can use as entry point every request that accesses a file, I choose to define a goal aiming at finding those entry points and perform the tests directly in the concretization phase of the framework.

Definition 20 (File Permissions)**.** *Let $x \in UserName$, $d \in Data$ be a data of the web application, the matrix M^0 the initial state, and M^i the matrix after the use of i actions. We can test if an attacker can have access to restricted files if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1 M^1\alpha_2 \ldots \alpha_i M^i$ such that:*

$$ShowFS \in M^i[x, x.d]$$

This is described by the Alloy goal:

```
assert FileAccess{
  no s : State | some d : Data-NoData | some x : User{
  s.granted = x.session &&
  d in s.showFS
  }
}
```

4.2.2 Application Logic Goals

The goals that I specified so far mainly focus on the generation of test cases for common and well know vulnerabilities. In this section, I discuss another important class of vulnerabilities that rely on the exploitation of logic flaws.

Logic flaws are usually not detected by the common penetration testing tools mostly because they are difficult to characterize. While “traditional” vulnerabilities (e.g. SQL-Injetion and XSS) have standardized definitions and have security requirements common in all web applications (e.g. to avoid XSS the web application has to sanitize all the user inputs), logic flaws violate business level rules (that may be unique for each web application) and they are extremely difficult to isolate and identify.

The detection of logic vulnerabilities requires real understanding of workflow and dataflow of the web application. Black box automatic tools cannot understand the architecture of a web application that employs many different technologies and resources. In my approach, in order to detect logic vulnerabilities, the security analyst has to define the security requirements that the application should comply with. These requirements may be very different for each web applications, for example, while in an e-commerce application it is important that one payment is tied to only one order, in a e-health web application we have to make sure that a subset of the documents are handled by an authorized user (i.e., doctors and not nurses).

Each transaction of the process (defined through workflow and dataflow) can be discovered and divided into steps by the security analyst. In order to test this type of flaw the penetration tester usually:

- tries functions is a different order,
- tries to bypass some key functionalities,
- tries to make the web application process the transaction incorrectly.

In [59], the authors divide the tests for logic flaw in two approaches.

Test for Fail-Open Conditions: For each function in which the application checks a user's credentials, including the login and password change functions, the security analyst should (i) walk through the process in the normal way, using an account the tester controls, (ii) note every request parameter submitted to the application, and in order to interfere with the application's logic, (iii) for each parameter, include the following changes:

- Submit an empty string as the value.
- Remove the name/value pair.
- Submit very long and very short values.
- Submit strings instead of numbers, and vice versa.
- Submit the same named parameter multiple times, with the same and different values.

If one modification causes a change in the web application behavior, the security analyst has to try to combine this with other changes to push the application's logic to its limits.

Test Any Multistage Mechanisms: If any authentication-related function involves submitting credentials in a series of different requests, the security analyst has to identify the apparent purpose of each distinct stage, and note the parameters submitted at each stage. She also has to repeat the process numerous times, modifying the sequence of requests in ways designed to interfere with the application's logic, including the following tests:

- Proceed through all stages, but in a different sequence than the one intended.
- Proceed directly to each stage in turn, and continue the normal sequence from there.
- Proceed through the normal sequence several times, skipping each stage in turn, and continuing the normal sequence from the next stage.

In order to use these two approaches during the tests, I show two possible examples of goals.

Check Payment

Let's assume that the security analyst is dealing with a web application for electronic commerce that permits to order and pay for the delivery of goods (i.e., the items sold by the organization). In an application like this, it is presumable to have a multi-stage mechanism that enforces the business logic that retrieves the needed information. In the following, I give an example of goal that produces attack traces that break the application logic in order to confirm an order without paying (i.e., entering the payment information) for the items.

Definition 21 (Check Payment)**.** *Let ConfirmOrder, Payment ∈ Data be data of the web application referring to an order confirmation and the payment information, $x \in$ UserName, the matrix M^0 the initial state, M^i and M^j matrices after the use of i and j actions with $i > j$. We can check if the payment can be avoided on the web application if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1 M^1\alpha_2 \dots \alpha_i M^i$ such that:*

$$\mathit{WriteDB} \in M^i[x, x.ConfirmOrder] \wedge \mathit{Checked} \notin M^J[x, x.Payment]$$

stating that the a user can send a request for an order even if the application does not check if the payment information are being provided.

As an example of goal instantiated in Alloy, the following is a direct translation of the general goal in the Alloy formalism for the case study discussed in Section 5.4.2:

```
assert CheckPayment {
        no s : State | some d : Payment {
        ConfirmOrder in s.action  &&
        s = last &&
        d not in s.checked
        }
}
```

Check Types

As an example of how to break the application logic, let's assume to be testing a web application where it is possible to transfer money from a user to another (other examples are possible, the money transfer has been used as an example of the logic attack in [59]).

Let the following action be the one defining the functionality that permits the transfer of the money from a user to another:

```
$MoneyTransfer(x,\ amount,\ y)$
  if  $M[x, x.session]\ =\ Granted$
  if  $x\ != \ Anon$
    Reset $M$ for $x$
    Add $Checked$ into $M[x, x.moneyTransfer]$
End
```

where *amount* has type *Int*. With actions like the one above, I can define security goals in order to raise a warning for possible flaws regarding the inserted amount. In the example, the amount transferred from the user x to the user y has to be checked for negative or non integer values.

Since this type of vulnerabilities can vary significantly regarding the modeled web application, the instantiation library used is crucial for testing the application, and the security analyst has to select a proper instantiation library for the different types of tests she wants to perform (as I have explained for the tests for "fail-open conditions").

The possible attack traces that have to be used in order to test the basic types are found via model checking with the search of those execution fragments that contain data referring to basic types.

Definition 22 (Check Types). *Let $x \in UserName$, $d \in Data$ be a data of the web application, the matrix M^0 the initial state, and M^i the matrix after the use of i actions. We can check the basic types of the web application if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1 M^1\alpha_2 \ldots \alpha_i M^i$ such that:*

$$WriteDB \in M^i[x, x.d] \wedge d \text{ has a basic type}$$

As an example, the goal can be instantiated in Alloy for the basic type "integer" (`int`):

```
assert CheckBasicTypes{
  no s : State | some d : Int | some x : User{
  s.granted = x.session &&
  d in s.writeDB
  }
}
```

4.2.3 Cross-Site Scripting

As I have discussed in Section 2.4.2, cross-site scripting attacks are a type of injection flaws, where malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.

Stored XSS

Stored attacks are those where the injected script is permanently stored on the target servers and the victim then retrieves the malicious script from the server when it requests the stored information.

Example 17 (Possible entry points for a Stored XSS). Typical examples of stored user input can be found in:

- Profile pages - since the data in the profiles are modifiable, they are one of the first target of the tests for stored XSS.
- Files - if the application allows the upload of files (and their visualization), a security analyst can create a file containing the payload and try to trigger the attack retrieving the file.
- Forum/Message boards - posted messages can be used to store XSS payloads.
- Logs - if the application maintain logs, payloads for XSS can be submitted to these logs.

△

Definition 23 (General Goal (one agent)). *The general goal for a stored XSS corresponds to the mapping of the possible entry points on the web application. Let* $x \in \mathit{UserName}$, $x.d \in \mathit{Data}$ *be a data of the web application,* $i, j \in \mathbb{N}$ *with* $i < j$ *the matrix* M^0 *the initial state,* M^i *and* M^j *the matrices after the use of* i *and* j *actions. A single user Stored XSS occurs on the web application if there is a state* M^i *at the end of an execution fragment*

$$\varrho = M^0\alpha_1 M^1\alpha_2 \dots \alpha_i M^i \alpha_{i+1} M^{i+1} \dots M^{j-1}\alpha_j M^j$$

such that:

$$\mathit{WriteX} \in M^i[x, x.d] \;\land\; \mathit{ShowX} \in M^j[x, x.d]$$

where

$$\mathit{WriteX} \in \{\mathit{WriteDB}, \mathit{WriteFS}, \mathit{WriteSD}\} \text{ and}$$
$$\mathit{ShowX} \in \{\mathit{ShowDB}, \mathit{ShowSD}, \mathit{ShowFS}\}\ .$$

Definition 24 (General Goal (two agents)). *Let* $i, j \in \mathbb{N}$ *with* $i < j$, $x, y \in \mathit{UserName}$ *with* $x \neq y$, $x.d \in \mathit{Data}$ *be a data of the web application, and* ϱ *an execution fragment. A multiple-users stored XSS occurs on the web application if there is a state* M^i *at the end of an execution fragment*

$$\varrho = M^0\alpha_1 M^1\alpha_2 \dots \alpha_i M^i \alpha_{i+1} M^{i+1} \dots M^{j-1}\alpha_j M^j$$

such that:

$$\mathit{WriteX} \in M^i[x, x.d] \;\land\; \mathit{ShowX} \in M^j[y, x.d]$$

where

$$\mathit{WriteX} \in \{\mathit{WriteDB}, \mathit{WriteFS}, \mathit{WriteSD}\} \text{ and}$$
$$\mathit{ShowX} \in \{\mathit{ShowDB}, \mathit{ShowSD}, \mathit{ShowFS}\}\ .$$

Specified Goal: On a database As an example of instance, a goal for stored XSS (with two agents involved) via a database is:

```
assert StoredXSS {
  no s : State | some d : Data-NoData |
  some s' : State| some x,y : User{
    d in s.writeDB &&
    s.granted = x.session &&
    d in s'.showDB&&
    s'.granted = y.session &&
    x != y &&
    lt[s, s']
  }
}
```

The goal is straightforward from the general goals above:

- A data has to be written on the database in a state *s* (`d in s.writeDB`) by a user with a valid session (`s.granted = x.session`).
- The same data has to be retrieved and showed from the database in a state *s′* (`d in s'.showDB`) by a user with a valid session (`s'.granted = y.session`).
- The users have to be distinct (`x != y`).
- The state where the data is written is antecedent to the one where it is showed (`lt[s, s']`).

The same goal can be rewritten in order to target the search of counterexamples (and thus the testing phase) to a subset of data. The following goal defines a stored XSS where only the profiles of the users can be used in order to deliver and trigger the payloads.

```
assert StoredXSSaimed {
  no s : State | some d : Profile |
  some s' : State| some x,y : User{
    d in s.writeDB &&
    s.granted = x.session &&
    d in s'.showDB&&
    s'.granted = y.session &&
    lt[s, s']
    }
}
```

Specified Goal: via File Upload As another example, a stored XSS vulnerability has to be tested also in those cases where the web application allows the upload and visualization of files.

As before, the general goal for stored XSS is simply translated in an Alloy goals describing the desired features:

```
assert XSS_FileUpload {
  no s : State | some d : Data-NoData |
  some s' : State| some x : User{
    s.granted = x.session &&
    d in s.writeFS &&
    s'.granted = x.session &&
    d in s'.showFS &&
    lt[s, s']
  }
}
```

Reflected XSS

Reflected XSS attacks are the subset of XSS attacks where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request.

General Goal Since the possible entry points for a XSS attack can be various, modeling server side technologies can be of help but the definition of a general goal remain difficult. A security analyst can be confident in the presence of this type of attacks only testing the web application providing data as a client and checking if these data are used by server-side scripts to parse and display a page of results.

Definition 25 (Reflected XSS)**.** *Let $x \in UserName$, $d \in Data$ be a data of the web application, the matrix M^0 the initial state, and M^i the matrix after the use of i actions. A XSS attack occurs if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1M^1\alpha_2\ldots\alpha_iM^i$ such that:*

- *d is used as parameter of α_i, and*
- *d is used by server-side scripts and rendered to a page of result(s), i.e.,*

$$M^i[x, x.d] \neq \emptyset \ \wedge \ ShowX \in M^i[x, x.d]$$

where

$$ShowX \in \{ShowDB, ShowSD, ShowFS\} \ .$$

From these premises, I give examples of specified goal for

- Reflected data - when a data, that is part of a request, is displayed as a duplicate, as part of a response, the security analyst can assume that some server-side scripts are used to parse the data.
- Checked data - when the security analyst has the confidence that the data are part of a request that is parsed on the web application, she should test for XSS attacks.

- URLs - sometimes the URL of a resource contains the data that are parsed in order to access the resource itself; in these cases XSS attack could be possible.

Specified Goal: Reflected data While modeling reflected data on the web application, the security analyst can use the testing-related information "*Echo*" (Section 3.5). A simple goal that can be used in order to find those traces that contain this type of data is the following:

```
assert XSS {
  no s : State | some d : Data-NoData | some x : User{
    s.granted = x.session &&
    d in s.echo &&
    NoData not in s.showDB
  }
}
```

Specified Goal: Checked data Those functionalities that uses the data provided by the users as part of a query can be used as entry points for XSS attacks. In this scenario, the web application could use the provided data as part of a script, and thus render the data triggering the XSS payloads.

The goal required for this type of analysis is the following:

```
assert XSScheck{
  no s : State | some d : Data-NoData | some x : User{
  s.granted = x.session &&
  d in s.checked &&
  NoData not in s.showDB
  }
}
```

The goal has been defined for the search for counterexamples with the following features:

- A user must have a valid session (`s.granted = x.session`),
- The data used in the action is part of a query (`d in s.checked`).
- Some results (that can be different from the initial data) have to be displayed (`NoData not in s.showDB`).

Specified Goal: XSS via URLs The last goal that I discuss for XSS is the one concerning the URLs of the resources of the web application. As I have explained in Section 3.5, when a page is read from the file system, rendered and displayed, the security analyst can insert in the model the testing-related information *PageIncluded*. Whenever this information is found in a model, the test for XSS should be performed. The execution fragments that permit this type of test can be found with the following goal:

```
assert urlXSS {
  no s : State | some d : FileAddress | some x : User{
    s.granted = x.session &&
    d in s.PageIncluded
  }
}
```

4.2.4 SQL-Injection

A SQL-Injection attack consists in the insertion or “injection” of SQL queries via the input data from the client to the application (see Section 2.4.2 for further details).

General Goal This type of attacks rely on the fact that SQL commands are injected into input in order to carry out the execution of predefined SQL commands. In order to differentiate those execution fragments where SQL commands can be injected, a security analyst can write a goal that searches for those execution fragments where a data is verified on the database and another is read (in this scenario, I restrict the range of possible web technologies, and assume the fact that some SQL query is involved).

Definition 26 (SQL-Injection). *Let $i \in \mathbb{N}$, $x \in UserName$, ϱ be an execution fragment and $x.d_1, x.d_2 \in Data$. A SQL-Injection occurs on the web application if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1M^1\alpha_2\ldots\alpha_iM^i$ such that:*

$$Checked \in M^i[x, x.d_1] \ \wedge \ ShowX \in M^i[x, x.d_2]$$

where $ShowX \in \{ShowDB, ShowSD, ShowFS\}$.

Specified Goal: On a Database The first goal that I show is a direct translation of the general goal in the Alloy formalism:

```
assert SQLInj {
  no s : State | some d : Data-NoData | some x : User{
    s.granted = x.session &&
    d in s.checked  &&
    NoData not in s.showDB
  }
}
```

where with `NoData not in s.showDB` I am stating that the set `showDB` has not to be empty (i.e., a data has been retrieved from the database).

Specified Goal: SQL-Injection via File System As for the XSS attack, also for SQL-Injection it is possible to deliver payloads via the file system. One instance is the case where the application maintains logs of some sort (where the request data is saved on the file system in order to be retrieved by the application administrator) where the SQL-Injection payloads can be submitted (an example can be found for DVWA in Section 5.3).

We can specify a security goal for the SQL-Injection via file system as the following:

```
assert SQLInjFileSystem {
  no s : State |  some d : Data-NoData |  some x : User{
    s.granted = x.session &&
    d in s.checked  &&
    (NoData not in s.showDB or NoData not in s.showFS)
  }
}
```

The goal extends the SQL-Injection entry points adding the possibility of delivery payloads in those cases where the data is used in a query that retrieve data from the file system.

String SQL-Injection for the login phase Another instance where a SQL-Injection can be possible is where a login functionality is implemented on the web application. The majority of the authentication mechanisms rely on databases for the storage of the credentials associated with the users. The credentials provided by a user during the authentication phase are checked along with those stored on the database by the web application. During the login an attacker can try to bypass the authentication mechanism with the injection of ad-hoc SQL statements.

Definition 27 (SQL-Injection for the login phase)**.** *Let $i \in \mathbb{N}$, $x \in UserName$, ϱ be an execution fragment. We can check if a SQL-Injection can be used to bypass a login phase if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1 M^1\alpha_2 \dots \alpha_i M^i$ such that:*

$$Checked \in M^i[x, x.Credential]$$

The goal is translated in the Alloy formalism as:

```
assert BypassLoginSQL {
  no s : State |some d : Credential   |some x : User{
    s.granted = x.session &&
    d in s.checked
    }
}
```

4.2.5 OS Commands

Command-Injection (Section 2.4.2) is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application.

General Goal When the security analyst supposes that the web application uses OS commands, as I have introduced in Section 3.5.2, she can insert in the model the assertion *Exec*. In order to derive the execution fragments containing this assertion the goal became a simple search of the states containing *Exec* for some data.

Definition 28 (OS Commands). *Let $i \in \mathbb{N}$, $x \in UserName$, ϱ be an execution fragment and $x.d \in Data$. I launch attacks related to a command execution if there is a state M^i at the end of an execution fragment $\varrho = M^0\alpha_1 M^1\alpha_2 \ldots \alpha_i M^i$ such that:*

$$Exec \in M^i[x, x.d]$$

The goal that can be used to search where to launch Command-Injection attacks is translated in the Alloy formalism as:

```
assert CommandExec {
  no s : State | some d : Data-NoData | some x : User{
    s.granted = x.session &&
    d in s.exec
  }
}
```

Instantiate Goal: Command-Injection on special functions In the previous goal, I have shown how the security analyst can target specific areas of the web application with a Command-Injection attacks inserting in the model the assertion *Exec*. Another entry point for this type of attacks sometimes can be found in those functionalities that retrieve files from the file system in order to display them. In this cases the security analyst can test the web application for possible Command-Injections with the following goal:

```
assert FileAccessCommandInjection{
  no s : State | some d : Data-NoData | some x : User{
    s.granted = x.session &&
        d in s.showFS &&
        d in s.exec
  }
}
```

4.2.6 AJAX Flaw

As discussed in Section 2.4.3, AJAX technologies are young and suffer from several security issues. A general definition for the flaws that concern AJAX technologies is not possible. In the following I give some examples that can be used during the testing of a web application.

Instantiate Goal: DOM-Injection One of the issues that I consider concerns those functionalities that are disabled in the Document Object Model (DOM). The security analyst can label these functionalities with the assertion *disabled* and thus target the tests during the concretization phase. The goal that I define for finding the execution fragments containing such functionalities is the following:

```
assert AJAXdisabled{
  no s : State | some d : Data-NoData | some x : User{
    s.granted = x.session &&
    d in s.AJAX &&
    d in s.disabled
  }
}
```

During the concretization phase the test execution engine has to consider the possible causes that are blocking the functionalities and try to bypass them.

Reflected XSS via AJAX AJAJ functionalities can be attacked with XSS attacks. As I have explained before, when a data (as part of a request) is displayed identical as part of a response, XSS should be tested. The goal to use in such cases is the following:

```
assert AJAXxss {
  no s : State | some d : Data-NoData | some x : User{
    s.granted = x.session &&
    d in s.AJAX &&
    d in s.echo
  }
}
```

The goal extends the one for XSS attacks (Section 4.2.3) in order to target AJAX functionalities. This extension is due to the fact that I use the goal name during the concretization phase of the framework in order to focus the testing phase (i.e., restrict the tests to AJAX technologies).

4.2.7 Brute Force

From a model-based testing perspective, brute force is not an attack that can be defined. This type of attacks can be used in various ways, and the data that are targeted can vary consequently. In this section, I show an instance of goal for a brute force attack that searches the state where the credentials of a user are used during the interaction with the web application. The aim of the goal is to derive those execution fragments where we can launch the attack and deal with the actual test during the concretization phase. Other aimed attacks are possible, but determining if they are successful can be done only in the actual testing phase.

Password Brute Force In the following goal, I assume that:

- The credentials are modeled through the signature `Credential`.
- The targets of the test are those entry points where the credentials are used in a query on the database (define by the assertion *Checked*).

These assumption can be instantiated in Alloy with the following goal:

```
assert PasswordBruteForce {
  no s : State | some d : Credential | some x : User{
    s.granted = x.session &&
    d in s.checked
  }
}
```

4.3 Concretization Methodology

In the following section, I illustrate the concretization methodology employed for the test of web applications. The methodology uses

- the abstract data contained the *Counterexample(s)* (resulting from the model checking phase),
- the *Configuration Values* needed for the correct interaction with the web application, 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 web applications).

In the framework these steps are depicted as ④, ⑤ and ⑥ in Figure 1.1.

4.3.1 Counterexamples

The output of the model checking phases is a set of *Counterexample(s)* (CEs) that violate(s) the goal. A CE is not directly executable on the web application, and I have to deal with the problem of filling the abstraction gap between the CEs and the web application's implementation. Every CE generated by the Alloy analyzer contains the sequence of actions used in order to violate the security goal and the set of states used (i.e., the instances of the matrix M).

As an example, the following are two instances of traces of actions that can be generated by Alloy:

```
[NoAction, Login, ListId, GetSearch, Search]

[NoAction, Login, ListId, ViewProfile, GetEdit,
 UpdateProfile, Logout, Login, ListId, ViewProfile]
```

Every trace starts with "`NoAction`" and ends with the action that permits to test the vulnerability (case above) or check if the attack can be triggered in the web application (case below). In order to differentiate these two cases from a the source code point of view, the `Skolem` constant (generated for each trace[2]) are used.

As an example, the skolem constant

```
[4, AliceName, Alice]
```

refers to an attack trace where the attack is triggered on the same location where the checks about its success can be made. More precisely, at state `4`, the web application can be attacked using the entry point `AliceName` and the agent `Alice`. Another example of skolem constant is:

```
[5, AliceProfile, 9, Alice, Bob]
```

where the web application is attacked at state `5` using `AliceProfile` as an entry point and `Alice` as user, and at state `9` the checks about the possibility of triggering the attack are made using agent `Bob`.

Other useful information that is contained in the counterexample, are the values of the set AP in the different states (I must recall here that in Section 4.1.4, I changed the concept of state in an orthogonal way with respect to the one presented in Section 3.6).

As an example, the following code refers to the evolution of the atomic propositions `WriteDB` and `ShowDB` through the different states of the transition system:

```
ShowDB  = [ ,                 , [BobId, AliceId], BobProfile]
CheckDB = [ , AliceCredential,                  , BobId]
```

Merging the information contained in these sets, I can have a vision on what is happening to the data handled by the web application. In the example, at state 1, the credentials of Alice are checked on the database, at state 2, the IDs of Alice and Bob are displayed, and at state 3, the ID of Bob is checked and his profile displayed.

[2]In Alloy, quantified formulas can be reduced to equivalent formulas without the use of quantifiers. This reduction is called skolemization and is based on the introduction of one or more skolem constants or functions that capture the constraint of the quantified formula in their values.

4.3.2 Configuration Values

The *Configuration Values* are those information that are needed in order to interact with the web application. In this section, I include as configuration values all the information that are needed in order to create the correct HTTP request(s) for each action regarding the web application and the low-level definition of the actions as used in the implementation of the framework (see Section 4.4). The notation used in this section thus reflects the python code used to implement these values.

The starting point for the interaction with the web application is a URL:

```
starting_URL =
     'http://127.0.0.1:8080/WebGoat/attack?Screen=71&menu=1100'
```

From this URL the actions that compose the attack trace are executed sequentially.

As I have discussed before, concatenated actions are possible. The security analyst can insert this information with the following code.

```
views = {'Login':'ListId'}
```

Since the data handled by the model can be abstract, it is not possible to define a direct relation between them and the actual data handled by the web application. In order to fill the gap between these two sets, different strategies are possible:

- Manually define the relation between these data.
- Ask to the security analyst the data to fill the needed fields for each HTTP requests that compose the counterexample.
- Harvest the values from the HTTP requests with a proxy (e.g., Wireshark) from a valid interaction of the security analyst with the web application.

In the implementation of the framework, I choose to define all the relations between real and abstract data. This choice (i) simplifies the implementation of the framework (since no complex modules are required), and (ii) gives a starting point for a future extension of the framework with the other possible strategies.

As an example, the code in Table 4.4 defines the relation between the abstract data handled by the model and the actual data handled by WebGoat.

For the readers familiar with python, the data structure used in Table 4.4 is implemented as a dictionary of dictionaries where for each user I store for each abstract data the real ones used by the web application.

The last set of concretization values is the one concerning the low-level definition of actions. In the following, I discuss two possible low-level definitions, one that uses Selenium and another that uses Python.

Table 4.4: Definition of the relation between the abstract data handled by the model and the actual data handled by WebGoat.

```
Data = {}
Data['Tom'] = {
  'Credential':{
    'employee_id':'105',
    'password':'tom'
    },
  'Profile':{
    'firstName'         :'Tom',
    'lastName'          :'Cat',
    'address1'          :'2211 HyperThread Rd.',
    'address2'          :'New York, NY',
    'phoneNumber'       :'443-599-0762',
    'startDate'         :'1011999',
    'ssn'               :'792-14-6364',
    'salary'            :'80000',
    'ccn'               :'5481360857968521',
    'ccnLimit'          :'80000',
    'description'       :'Co-Owner.',
    'manager'           :'105',
    'disciplinaryNotes':'NA',
    'disciplinaryDate' :'0',
    'employee_id'       :'105',
    'title'             :'Engineer'
    },
  'Name' : {'search_name' :'Tom'},
  'Id':{
    'employee_id':'105'
    }
}
```

Selenium The first methodology uses Selenium [55] for the definition of actions as HTTP requests. Selenium is a Firefox plugin that allows one to record, edit, and debug tests. Selenium is allowed to record only actions that can be performed through a web browser (e.g., clicking a button, selecting from a drop-down menu).

The Selenium features that I use are (i) the recording of browser actions in order to generate HTTP requests and (ii) the possibility to replicate these actions. The concretization phase relies on the fact that the security analyst can record, through Selenium, the set of browser actions that compose each action. Two examples of actions recorded with Selenium are:

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

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

Using the browser actions also means to limit the investigation of possible vulnerabilities only to the browser interface of the web application (i.e., I can not change the HTTP requests that the user created by the user interface of the web application).

Python The second methodology for the concretization of actions (and the one used in the implementation of the framework), uses the python engine in order to build HTTP requests. Since the data composing the HTTP request are already present in the configuration values, in order to differentiate the different actions I have to distinguish them as form actions (i.e., the ones generated by HTML forms), functionalities, web pages, and so on. I clarify this concept with the examples following below.

Example 18. These actions refer to HTML forms and are appended to the URL of the page where the action as to be performed:

```
actions = {}
actionsURL = {
  'NoAction'     : '',
  'Login'        : '&action=Login',
  'Logout'       : '$action=Logout',
  'ViewProfile'  : '&action=ViewProfile',
  ...
}
```

These actions refer to PHP pages (the same can be done with similar technologies such as ASP):

```
actionsURL = {
  'NoAction': '',
  'Login': 'index.php',
  'SelecProfile':'dashboard.php',
  'ViewProfile': 'profile.php',
  ...
}
```

These actions refer to different technologies:

- form actions,
- functionalities implemented on the web application by special functions (with or without fixed parameters), and
- HTML pages.

```
actionsURL = {
'NoAction': '',
'Login': 'login',
'AddMessage':'newsnippet.gtl',
'ShowMessageAsUser':'snippets.gtl',
'ShowMessage':'snippets.gtl?uid=caio',
'ViewProfile': 'editprofile.gtl',
'UpdateProfile': 'saveprofile?action=update&',
'Logout':'logout',
'FileUpload': 'upload.gtl',
'FileAccess':'caio/FileUploadXSS.html',
'Refresh': 'feed.gtl?'
}
```

△

4.3.3 Instantiation Library

The *Instantiation Library* (InstLib) contains data such as attack strings (e.g., payloads for XSS), common malicious input (e.g., a set of passwords for a brute force attack) and scripts to be used as test patterns (i.e., script to be executed client-side in order to test the web application).

In addition to the payloads used during the attack, in the InstLib, I also specify the expected result that the security analyst expects for each payload. As an example, the following list of passwords can be used for a brute force attack:

```
P_B_F=[
  ["admin",''],    ["test",''],     ["administrator",''],
  ["john",''],     ["12345",''],    ["asd",''],
  ["qwerty",''],   ["qwertz",''],   ["watson",''],
  ["Password",''], ["password",''], ["tom",'']
]
```

Since different web applications can react differently after a successful login (i.e., the success of the brute force attack is not related to the payload used), I separate this feature of the framework and create a list of success criteria such as the following:

```
P_B_F_CheckThis=[
  '*bash*',
  'My Snippets',
  'Alice <Alice>',
  'Profile',
  'Welcome Back',
  'Welcome to the password
     protected area admin'
]
```

This list is not meant to be exhaustive for all the possible web applications. A security analyst can insert the success criteria that are best suited for the web application she is testing. In order to do so, the security analyst has to detect on the target web application which information discriminate a successful login from an unsuccessful one.

Another example of InstLib is the one for XSS attacks:

```
S_XSS=[
  ['alert(String.fromCharCode(88,83,83,65,116,116,65,99,107))',
   'XSSAttAck'],
  ['<onmouseover="alert(1)"href="#">readthis!</a>',
   'readthis!'],
  ['alert("XsSatt");',
   'alert("XsSatt");'],
  ['<script>alert("XSSatt");</script>',
   '<script>alert("XSSatt");</script>'],
  ['<script>alert("veryDangerous");</script>',
   'veryDangerous'],
  ["red'onload='alert(1)'onmouseover='alert(2)",
   'alert(1)']
]
```

In this case the expected values are related to the payloads that can generate them and thus are saved in the InstLib as couples.

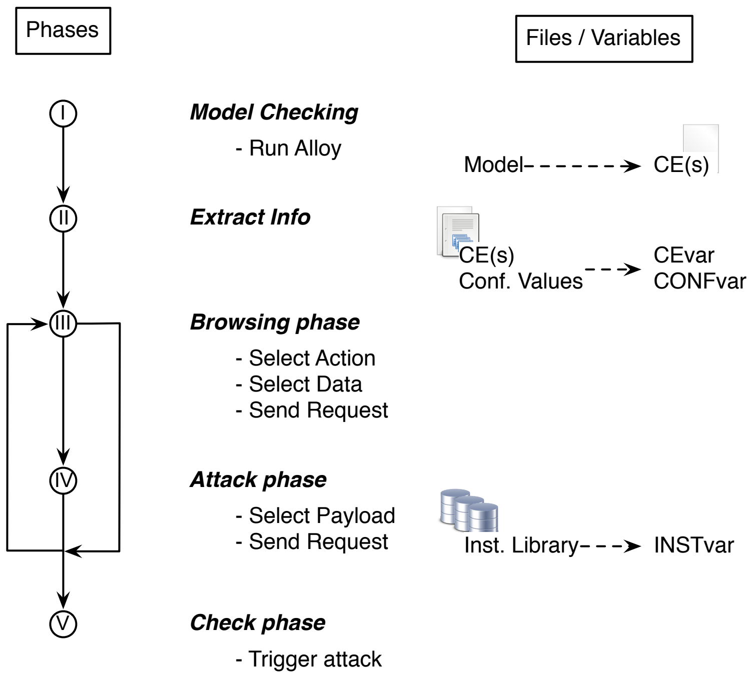


Figure 4.1: Workflow and data used in the implementation of the framework.

4.4 The Implementation of the Framework

As an initial proof of concept, I have developed a preliminary version of a tool written in python. The complexity of the code does not permit a full explanation of the different modules that compose the tool, thus I present the execution workflow (depicted in Figure 4.1) of the tool.

Phase I: Model checking In this phase the tool automatically runs the Alloy analyzer on a prewritten model of a web application. During this phase, I assume that the model contains the specification of web application along with the security goal that has to be tested. If a counterexample is found, it is saved in a text file containing all the information about the transition system's states. This phase is automatically executed by the python engine and thus the security analyst has only to define the model with the goal.

Phase II: Information extraction If a counterexample has been found during the first phase, the file containing the information about the CEs is parsed, along with the configuration file, in order to populate the python variables that will be used in the next phases.[3] The extraction of these data

[3]In order to maintain a limited number of files used by the security analyst, in the implementation of the framework the Configuration Files and the Low-Level definition of the actions are merged in the same file.

takes into consideration the scenario where multiple traces (i.e., multiple counterexamples) are generated for the same goal. The data structure in python reflects the number of traces present in the counterexample(s).

Phase III: Browsing Having all the information needed for the interaction with the web application, this phase deals with the problem of reaching the exact location where the attack has to be made. Knowing from the CE at which state the attack has to be made (let us assume at state 5), the python engine selects, one by one, the actions from the first to the one before the attack (i.e., 4). For each action, the engine checks all the data used in the model (i.e., the data type used during the modeling phase) and selects the proper data from the ones available in its data structure with regard to the possible input on the page (i.e., the possible HTTP requests that can be made). 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. Once the correct data has been selected a HTTP request is sent to the web application and the result retrieved.

Phase IV: Attack Once the browsing phase has been completed, the python engine switches to the attack phase. In the implementation of the framework I am not interested on complex implementation of this phase, and I choose to force the attacks hardcoding the locations for each case study; this particular choice reflects the possibility of using the VERA tool (see Section 4.5) as an automatic vulnerability testing tool during this phase. I should anticipate to the readers that, from the source code point of view, all the results obtainable with the VERA tool can also be obtained with the implementation of a complex python module that performs all the attacks contained in the VERA low-level attacker models. As will become evident in the next section, this scenario requires a complex maintenance of the tool once a testing methodology for a new attack/vulnerability has to be coded (i.e., the routines that tests a particular attack).

Phase V: Check After the delivery of every payload in the previous phase, a check phase starts. If the information contained in the CE require that the check for an attack has to be made in a different location of the web application (i.e., the success of the attack is not verifiable in the received HTTP response), an additional browsing phase (III) is called; if not (or after the end of the browsing phase) the HTTP response from the web application is checked in order to find the confirmation of the attack.

4.5 VERA tool

In this section, I present the VERA tool [12][4] standing for "VERA Executes the Right Attacks", which allows testers to define attacker models by means of extended finite state machines (EFSM). In this way, testers can define new tests where the payloads and the behavior are cleanly separated and that abstract away from low-level implementation details such as HTTP requests.

Testers are often confronted with situations where existing tools are of little help because (i) they do not account for a particular configuration of the System Under Test (SUT) or (ii) they do not include tests for certain vulnerabilities. For instance, the SUT could use a particular authentication mechanism (such as a proprietary protocol) and a recent/rare database version. It is likely that most available tools will not cover the authentication mechanism and might not have information about known vulnerabilities of the database model used by the SUT.

In this situation the tester would benefit from extension mechanisms to the available tools. Some tools (like the non-free version of Burp [31]) allow one to write such extension plug-ins. These plug-ins must be written in the programming language of the tool and imply the learning curve of the tool's API. The alternative consists in directly writing scripts for the task at hand.

4.5.1 Modeling

In the following I give brief presentation about the models that can be executed using VERA, as originally described in [57]. Here the goal is to illustrate the precise meaning of VERA models and not to perform formal reasoning on them. Attacker models can be seen as an extension of Mealy machines [35] with guarded transitions and variables. Similarly, one can see similarities between the attacker models and UML statecharts. The formal syntax and semantics given below however clarifies the differences between attacker models and Mealy machines or UML statecharts (there are many different semantics for UML statecharts; e.g. see [20, 64]). In particular, attacker models do not have a notion of composition.

Example

To begin, we illustrate the semantics by using the example in Figure 4.2. In the first transition the attacker starts the interaction with the system by sending a message requesting a particular URL. Subsequently, a message is received by the attacker that does not trigger an immediate reaction, but only a change in the configuration. With the new configuration (the new

[4]The VERA tool has beed developed during the SPaCIoS project, I mainly worked on the graphical user interface.

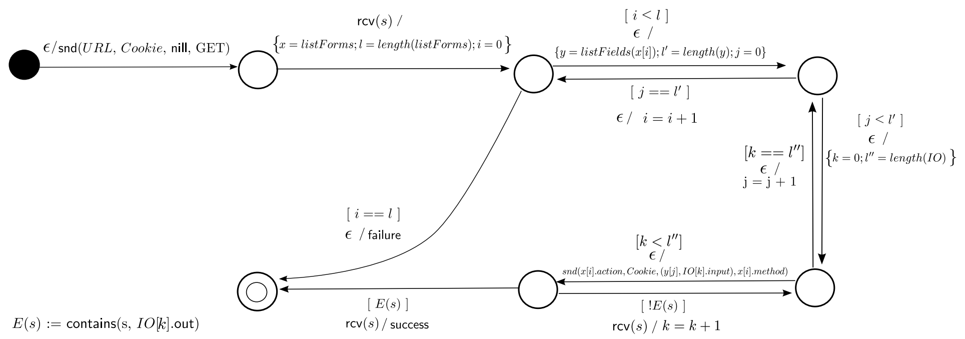

ϵ/snd(URL, $Cookie$, nill, GET)
rcv(s) /
$\{x = listForms; l = length(listForms); i = 0\}$
$[\ i < l\ ]$
ϵ /
$\{y = listFields(x[i]); l' = length(y); j = 0\}$
$[\ j == l'\ ]$
ϵ / $i = i + 1$
$[\ j < l'\ ]$
ϵ /
$\{k = 0; l'' = length(IO)\}$
$[k == l'']$
ϵ /
$j = j + 1$
$[\ i == l\ ]$
ϵ / failure
$[k < l'']$
ϵ /
$snd(x[i].action, Cookie, (y[j], IO[k].input), x[i].method)$
$[\ E(s)\ ]$
rcv(s) / success
$[\ !E(s)\ ]$
rcv(s) / $k = k + 1$
$E(s)$:= contains(s, $IO[k]$.out)

Figure 4.2: General injection low-level attacker model.

values of x, l and i) we have two possible transitions: (i) to the final state and (ii) the one to the third state.

The transition to the final state is triggered when there are no more fields to check (or the page pointed by URL doesn't contain any fields), and the attacker gives up by outputting failure. The two transitions, outgoing and ingoing, to and from the state represent a configuration change for the variables j, l' (for the outgoing vertex) and i (for the ingoing vertex).

In the lower-right state the attacker sends a message with the j-th payload in IO (to be delivered to the i-th field in the page) without receiving a message.

Subsequently the attacker receives a message and changes his state (thus he reacts) according to its content. If the message doesn't contain the expected output the state will change in order to check the next payload or the next field; on the other hand if the attacker receives the expected output he has found a vulnerability thus outputting success.

Formal syntax and semantics for attacker models that can be executed using VERA can be found in [12].

Implementation

The VERA tool is an extensible framework based on the concept of extended finite state machines that allows for the creation and execution of attacker models targeting generic vulnerabilities of web applications in a black-box fashion, in essence reproducing a penetration testers actions. These attacker models can be collected in libraries targeting specific vulnerability types across multiple types of web applications. Besides allowing semi-automatic online testing using the provided libraries, VERA can also be used in fully automated scripts by interfacing directly with the provided back-end or importing and using the libraries provided by the framework.

The back-end of the VERA framework consists of a python program which parses:

1. Selected **Instantiation library** containing data values used to interact with SUT.
2. **Configuration file** containing system specific information needed to test a SUT.
3. **Model file** An XML file containing the attacker model to be tested.

Then the framework creates an instance of the attacker model, and runs it online against the SUT by using the values from the three files and the user if needed.

Instantiation library

While the basic functionality of an attack is encapsulated within the attacker models described above, an actual attack quite often needs additional data that is ill suited to automata. An example for this would be a list of passwords in a brute force attack. A decision was taken to outsource this type of information, which is not application specific, into standardized instantiation libraries. These can be accessed by the attacker model as arrays, basically instantiating the attacks with specific values.

If an attack, or parts of an attack, can be performed multiple times with different values, in order to test whether one of these values triggers a vulnerability, these values should be moved into an instantiation file. This file can then be extended by the different security experts.

In some cases it makes sense to use multiple instantiation libraries, which can be used by the security expert in different circumstances. We have identified two common scenarios where the use of multiple instantiation libraries helps the security expert during a vulnerability assessment:

- If the same steps can be performed to create completely different attacks, then it makes sense to use different instantiation libraries for these kinds of attacks. An example for this would be the use of injection attacks (e.g. SQL-, X-Path-, Command-Injection). It makes sense to not have a single big library, but rather have different instantiation libraries. An example for this would be the use of Command-Injection attacks: While the attack itself, injecting some kind of code into HTTP requests, is the same, the actual values injected can target a wide variety of attacks. In order to perform tightly focused attacks, it makes sense to not have a single big library, but rather have different instantiation libraries for JavaScript-Cross-Site-Scripting attacks, SQL-Injection, X-Path Injection, Command-Line-Injection, etc.
- If there are a large number of values, and these can be divided based on information which might be available during the test time, such as the back-ends used, it might make sense to split the instantiation

library. An example for this would be file enumeration, where different instantiation libraries might exist for the different kinds of platforms available. This can help an analyst perform targeted attacks as well as improve the efficiency of testing. If no information is available during the test, all instantiation libraries can be used, resulting in the same functionality as without such a split.

The VERA framework allows a user to run an instance of the attacker model with an instantiation library, though the GUI allows the user to select a number of instantiation libraries which are then run sequentially. This instantiation library consists of a simple text file containing an array called `IO` of either single values or tuples. Which kind of data is in the array depends largely on the attacker model it was created for. An example instantiation library for file enumeration is:

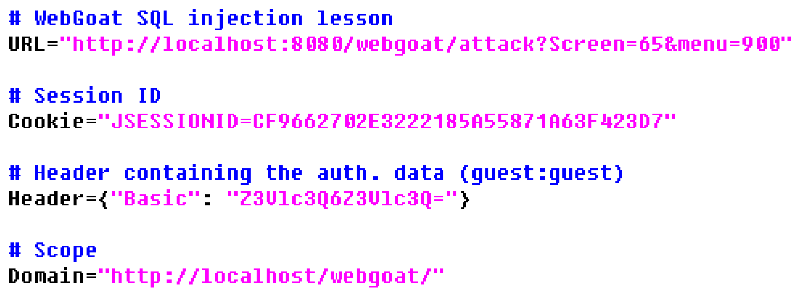
```
IO = [
   .htaccess,
   .htaccess.bak,
   .htpasswd,
   .meta,
   .web,
   conf,
   apache/logs/access.log ,
   apache/logs/access_log,
   apache/logs/error.log ,
   apache/logs/error_log,
   access_log,
   cgi
]
```

Configuration values

While the goal of the attacker model introduced is to be as generic as possible, once the testing has to be performed, the use of system specific knowledge is unavoidable. This information is stored in special configuration files which specify a number of variables and their values in the following format: `Name=Value`. Comments can be added using the `#` sign. These have to be assigned by the security expert before the tests can be performed.

For convenience, we have defined a number of parameters that should be used by all models, allowing the expert to create a single file containing all necessary values and then run all selected attacker models and their instantiation libraries using that same file:

- **URL** – This contains the target URL for the attacker model. Depending on the attacker model, this URL might just be used as a starting point used to crawl through the entire site.



```
# WebGoat SQL injection lesson
URL="http://localhost:8080/webgoat/attack?Screen=65&menu=900"

# Session ID
Cookie="JSESSIONID=CF9662702E3222185A55871A63F423D7"

# Header containing the auth. data (guest:guest)
Header={"Basic": "Z3Vlc3Q6Z3Vlc3Q="}

# Scope
Domain="http://localhost/webgoat/"
```

Figure 4.3: Configuration file for the WebGoat SQL-Injection lesson.

- **Cookie** – This contains the cookies necessary for the web application, such as a session ID.
- **Header** – If additional headers are needed for the correct functioning of the web application, such as authentication headers, they can be defined in this variable.
- **Domain** – The argument for this is a URL restricting the scope of the instantiation library. Only sites within the scope should be targeted by the attacker model.

Creators of new attacker models are strongly encouraged to use these default parameters, and refrain from extending the list unnecessarily, as this would complicate the interoperability of the different attack models used. If additional information is required the user should be prompted.

An example configuration file targeting the SQL-Injection lesson from WebGoat can be seen in Figure 4.3.

Attacker model

The attacker model has been implemented following the data structure presented in Section 4.5.1. The model is saved as an XML that uses the following tags:

```
<statechart> </statechart>
```

delimiting the parsed attacker model,

```
<node id="start"/>
```

a node in the diagram with its identifier,

```
<transition from="start"
            guard="True"
            input=""
            output=""
            to="send"> </transition>
```

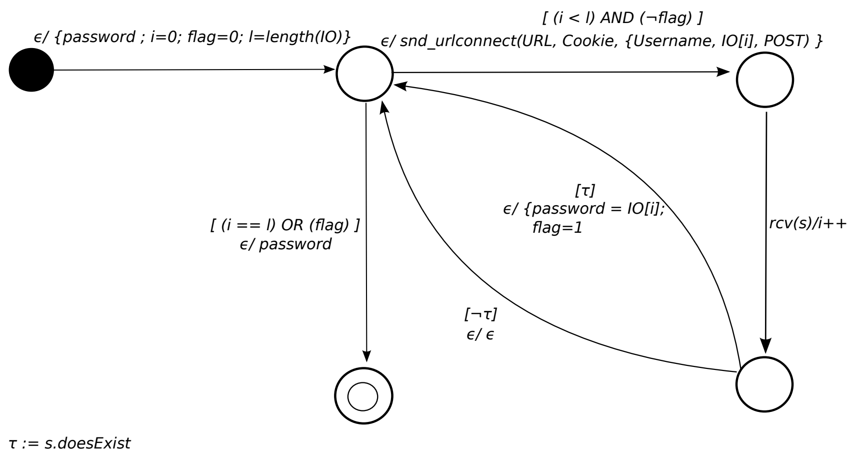


Figure 4.4: Low-lever attacker model for a brute-force attack.

the transition between two nodes with the corresponding values,

```
<action value="files=[]"/>
```

the actions to be performed during a transition.

In the graphical editor the information needed for the correct execution are saved in a SCM file (State Chart Model file).

Interfaces

Currently there are two possible ways to use the VERA tool. On the one hand, it is possible to directly call the command line backend written in python. This allows experienced users to use the VERA framework in various ways, for example by integrating it in more complex scripts, parsing the output using command line utilities or creating automated security scans at certain times. On the other hand, I created a graphical user interface which uses Eclipse as a back end. This graphical interface guides the user through the different options and provides the security analyst with an overview of the attacker models as well as the results.

4.5.2 Examples of Low-level Attacker Models

In the following I show two examples of low-level attacker models from [58].

Password Brute Force

A low level attacker model to perform a password brute-force attack (Section 2.4.4) is depicted in Figure 4.4.

The password brute-force attack model iteratively tests the login password for user accounts against a list of predictable values. This is achieved by sending corresponding HTTP requests iteratively to the application based on the given instantiation library. The HTTP response received from the application is analyzed to find the correct password to the user account.

The instantiation library consists of a list of common/predictable password values for user accounts. This list serves as an input for the execution of the model. For specific executions, the list can be adjusted to incorporate more or context specific values as per requirement.

The password brute force attacker model uses the following values from the configuration file to perform the automated attack:

- URL: Contains the target application URL
- Header: HTTP Request Header *Content-Type*
- Relative URL: Redirection URL on a successful login
- Form fields: Provide the name and value of the username field for the account to be tested, and additionally the name of the password field within the login form

The attacker model iterates over the instantiation list. For each password value within the list, it builds a valid HTTP Request and sends it to the vulnerable application. Apart from the URL, Headers values, etc., each HTTP request consists of the username of the user account supplied in the configuration file, and the password value taken from the instantiation list for the current iteration. The model waits for the HTTP response and analyzes it for the validity of the password that was sent in the HTTP request. The model checks and verifies the HTTP Status code sent in the response. If the HTTP response reveals a redirection to a valid user account page, the current password is marked as the valid password for the user account. Otherwise, the model continues to iterate against the list until either a valid password is found or the list has been exhausted.

Path Traversal

A low level attacker model to perform a path traversal attack (Section 2.4.1) is depicted in Figure 4.5.

In the first step, the path traversal attack model executes the necessary steps to gather all the input fields available within the login web form accessible at the given URL. As the next step, the model fixes the known field's value to the value provided in the configuration file and iteratively tests the targeted field's value against the instantiation list. As an example, given a web form with the "username" and "password" fields, the model fixes the value of the "username" field and performs an iterative test to find out the correct value of the "password" field against the instantiation list. This is achieved by sending corresponding HTTP requests iteratively to the

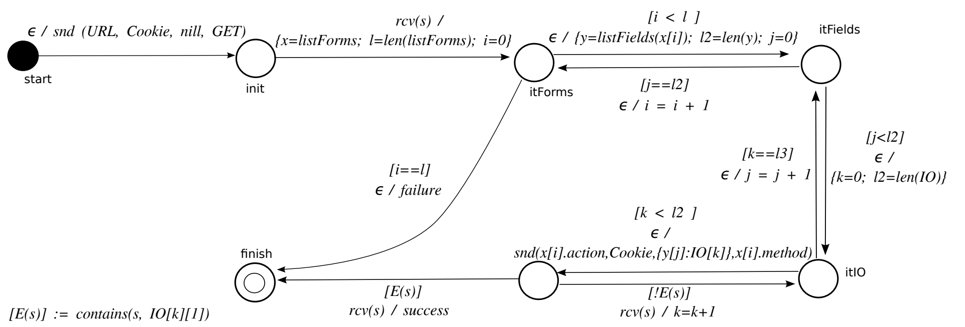

ϵ / snd (URL, Cookie, nill, GET)
start
init
rcv(s) / {x=listForms; l=len(listForms); i=0}
itForms
[i < l] ϵ / {y=listFields(x[i]); l2=len(y); j=0}
itFields
[j==l2] ϵ / i = i + 1
[j<l2] ϵ / {k=0; l2=len(IO)}
[k==l3] ϵ / j = j + 1
[i==l] ϵ / failure
[k < l2] ϵ / snd(x[i].action,Cookie,{y[j]:IO[k]},x[i].method)
itIO
finish
[E(s)] rcv(s) / success
[!E(s)] rcv(s) / k=k+1
[E(s)] := contains(s, IO[k][1])

Figure 4.5: Low-lever attacker model for a path traversal attack.

application for every value provided within the instantiation library. The HTTP response received from the application is analyzed to find whether the traversal and thereby the file operation was successful.

The instantiation library consists of a list of attack vectors that can be used to modify the meaning of the path and hence let the attacker break out of the web root.

4.5.3 Using VERA for Vulnerability Testing

The possible use of the VERA tool in the scope of the framework, is as an automatic test execution engine for the attacks that can be found via the model checking phase. The introduction of the VERA tool for the vulnerability testing brings along various advantages and make the tool itself coping with small problems it can suffer during the testing of complex attacks in web applications.

How can the tool be embedded in the framework? The VERA tool needs some parameters, defined in a configuration file (e.g., *URL,Cookie, Header*, and *Domain*), in order to correctly interact with the web application during the test. These parameters can be instantiated real time with the values available in the data structure of the python engine already implemented (even though it has been developed as an initial proof of concept).

Why should I use it? With the use of the VERA tool, one can keep separated the definition of the attacks (i.e., the low-level attacker models) and the parts of the framework that deal with (i) finding the attacks, (ii) "preparing" the web application to be attacked (i.e., the browsing phase), and (iii) controlling if the attack has succeeded (i.e., the check phase). With this setting the overall modularity of the framework brings the liberty of introducing new methodologies of attack (in general new features) without rewriting large parts of the code.

Are the low level attacker models reusable? Some adjustments are required on both the low-level attacker models and the python engine in order to permit the correct functioning of the VERA tool. However a security analyst can reuse the modified low-level attacker models outside the framework (i.e., in the case she wants to perform only a vulnerability testing). It is a matter of inserting in the already available models the information about the correct exchange of messages with the python engine.

Why not using only VERA? As an example, the cases where the attack is triggered on a different part of the web application are not covered by the VERA tool in its low-level attacker models that I have developed so far. Introducing this feature in the models bring a loss in their generality, can be not trivial, and brings a lack of reusability that could result in a new definition of every model for every web application that the security analyst wants to test.

4.6 Small Conclusion

In this chapter, I have presented how to fill the abstraction gap between the abstract tests (resulting from the model checking phase) and the actual implementation of web applications. The concretization methodology starts with the definition of the model in Alloy, and through the implementation of the framework and the low-level definition of the actions permits to perform tests on real web applications. In the following chapter, I show how the framework can be applied to four case studies.

Chapter 5

Case Studies

In this chapter, I discuss the application of the framework to the four case studies WebGoat, DVWA, Gruyere and OnlineShop. Since the tests have been performed with the preliminary implementation of the framework, I aim to show the range of tests that can be performed with this approach rather than an analysis of the performances of the tests (e.g., duration in time, number of requests, etc.).

In order to show the capabilities of the framework, I use three security tools as benchmarks for the various tests:

- Burp suite [31] (free version 1.3.03),
- OWASP Zed Attack Proxy [40] (ZAP, version 2.3.1), and
- Paros [17] (version 3.2.13).

These three tools are mainly proxy tools used to intercept and analyze the HTTP traffic from and to a web application, but they also provide some basic vulnerability scanning techniques that we employed for the tests. I am aware of the fact that they are not the most powerful tools for performing vulnerability scanning, but still they are the main general-purpose free alternatives currently available. Regarding the Burp suite, during the tests I have used the *intruder* functionality since the free version does not allow the use of the *analyzer*.

5.1 WebGoat

In this section, I show the definition of the model, the results of the model checking phase and the tests for the WebGoat case study.

WebGoat [41] is a deliberately insecure web application maintained by the Open Web Application Security Project (OWASP) [38] designed to teach web application security lessons. In each lesson, users must demonstrate

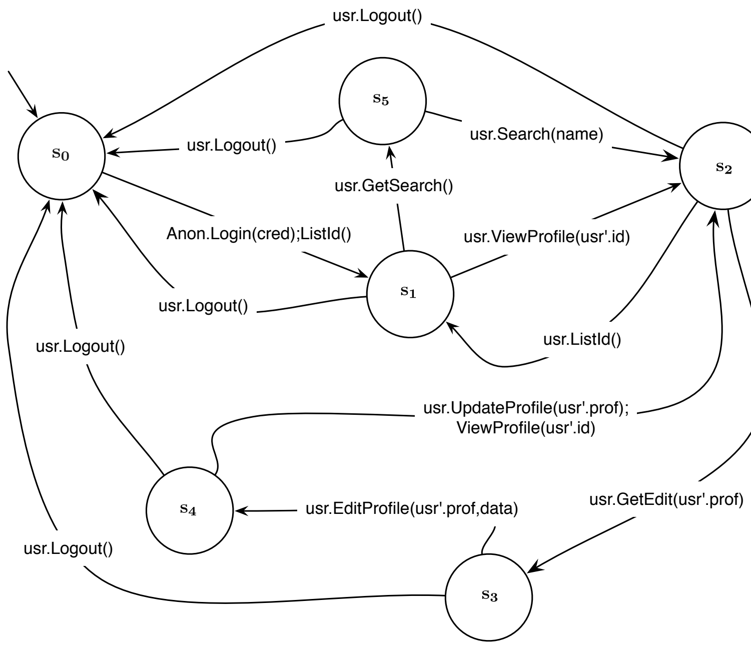


Figure 5.1: WebGoat high-level model.

their understanding of a security issue by exploiting a real vulnerability. For example, in one of the lessons the user must use SQL-Injection to steal fake credit card numbers. The application is a realistic teaching environment, providing users with hints and code to further explain the lesson.

The primary goal of the WebGoat project is simple: create a de-facto interactive teaching environment for web application security. In the future, the project team hopes to extend WebGoat into becoming a security benchmarking platform and a Java-based Web site Honeypot.[1]

5.1.1 General Model: WebGoat

The general model of WebGoat (depicted in Figure 5.1) refers to the lessons where the user can interact through a graphical interface that permits one to:

- Login to a restricted area,
- display and modify his profile,
- display (and for a subset of users, modify) other users' profiles, and

[1]A honeypot is a trap set to detect and deflect attacks to an information systems. It consist of a computer, data, or a network site that resemble a legitimate part of the network (i.e., it seems to contain information or resources of value to attackers), but is actually isolated and monitored.

Table 5.1: Definition of the actions for the WebGoat case study.

```
Login(x, x.credential)
 if M[Anon, Anon.session] = Granted
 if M[x, x.credential] = Initial
  Reset M for x
  Del Granted into M[Anon, Anon.session]
  Add Granted into M[x, x.session]
  Add Checked into M[x, x.credential]
End
```

```
ListId(x)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, y.id] = Initial ||
    M[x, y.id] = Gained
  Reset M for x
  Add ShowDB into M[x, y.id]
End
```

```
ViewProfile(x, y.id)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, y.id] = ShowDB
  Reset M for x
  Add ShowDB into M[x, y.profile]
End
```

```
GetEdit(x, x.prof)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, y.profile] = ShowDB
  Reset M for x
  Add Edit into M[x, x.profile]
End
```

```
GetSearch(x)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, y.name] = Initial ||
    M[x, y.name] = Gained
  Reset M for x
  Add Edit into M[x, y.name]
End
```

```
Search(x, y.name)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, y.name] = Edit
  Reset M for x
  Del Edit into M[x, y.name]
  Add Checked into M[x, y.name]
  Add ShowDB into M[x, y.profile]
End
```

```
UpdateProfile(x, x.prof)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, x.profile] = Edit
  Reset M for x
  Add WriteDB into M[x, x.name]
End
```

```
Logout(x)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Del Granted into M[x, x.session]
  Add Granted into M[Anon, Anon.session]
End
```

- search other users' profiles.

The tuple that contains the data used in the model is:

```
UserData =
   (
   cred,
   id,
   prof = (name, ...),
   session,
   )
```

stating that each user has some credentials (modeled as an unique entity), one identifier, a profile (where the only field needed in the model is the name), and a session with the web application (granted through the login phase).

It is interesting to note that this model gives the means to show two possible ways of modeling the same functionality (i.e., the *ListId*). In the specific, while defining the actions for the listing of users (i.e., their IDs), I can have two possible scenarios:

- all the users can see the other user IDs, or
- the users see only a subset of the available IDs.

In the first case, the modeling of such functionality is straightforward since the action shows, to the requesting user, all the IDs. In the latter, a security analyst introduces the known IDs in the initial knowledge of each user and shows them every time a *ListId* action is used. The second scenario has been used for the instantiation of the actions that I report in Table 5.1.

Even though this model can be used during different tests, some of the lessons reported in the following do not use it. This is due to the fact that these lessons contains only one functionality (modeled as a single action), and thus are not interesting for what concerns the possible interaction with the web application but only for the testing capabilities of the framework.

In the following:

- I refer to "lesson" as a subset of the pages of WebGoat (indicated in the title of the section).
- I discuss the counterexample for those `check` statements with the minimum number of states.
- In the result section I discuss
  - the success (✓) of the tests, i.e., if the tests have been able to find the vulnerability,
  - the causes of their failure (X),
  - the problems that make the test ineffective (~), and
  - if the tests are inapplicable (represented with NA).

5.1.2 Bypass a Path Based Access Control Scheme

Model and Goal

The model of this lesson is composed by a single action that permits one to display a file from a list of available files.

Since the model is simple the initial state instantiates a session to a guest user (the one that WebGoat uses for the real users that use the platform):

	Guest.session
Guest	*Granted*

The goal of the lesson is to access a resource that is not in the list given by the lesson (the attack is similar to the File Inclusion attack in DVWA 5.3). In order to perform the model checking phase I use the `FileAccess` goal introduced with the ones for File Permissions among the goals of Section 4.2.1 Page 70.

Attack trace(s)

The first `check` statement that returns a counterexample is:

```
check FileAccess for 2 State, 1 User, 2 Data
```

where I test the goal searching for counterexamples with 2 states, using one user and 2 data (i.e., the file address and the session). The resulting counterexample and skolem constant are:

$\texttt{Trc}_0$`=[NoAction, FileAccess]`
$\texttt{Sko}_0$`=[1, GuestFileAddress, Guest]`

In Trc_0 the only available action is used and the skolem constant express that at state 1 I can attack the application using the `GuestFileAddress` data and using `Guest` as user.

Concretization parameters

As concretization parameters, for this lesson I used as payloads a list of resources containing (among the others) the payload `../../main.jsp` (i.e., the one that complete the lesson). In order to check the correct completion of the lesson, the checking phase of the framework has been carried out with the check statement `Lesson completed` (i.e., the message displayed by WebGoat once a lesson has been solved).

Results and comparison with other tools

The results of the tests performed with the framework and with the other tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	✓	X

Paros was the only tool that was not able of solving the lesson (i.e., not able to find the vulnerability), while the others completed it without any problem.

Similar results are obtainable with ad-hoc scripts that try to retrieve the resources described in the payloads. Such testing methodology targets one entry point at a time and thus has to be relaunched for every entry point of the web application under test. The use of the framework permits the automatic selection of such entry point and their subsequent test without the need for a security analyst to relaunch the test.

Small conclusion

The main objection that I see in this lesson is the simplicity of the model (the same can be said for the lessons with the same structure that I will introduce later in this chapter). Having only one action in the model narrows the search space of the model checker to a singe path, and result in the forced choice of the action to be performed. I believe that the modeling choice of having a single action is supported by the fact of wanting to show the testing capability of the framework rather than the capabilities of the model checker (even though the same results can be obtainable with the analysis of more complex models).

5.1.3 Role Based Access Control: Bypass Presentational Layer Access Control

Model and Goal

The model used in this lesson is the one presented in Section 5.1.1. In order to differentiate the different users, the general model has been extended with a signature used in order to differentiate the administrator (`isAdmin`). This signature is checked in the conditions of the action `DeleteProfile` in order to check if it can be performed (or not) by the users.

The goal of the lesson is to access the "*Delete*" function as a regular employee "Tom" (the employee do not see a button with this functionality in his interface) exploiting a weak access control.

The initial state of the transition system is:

	Anon	*Tom*	*Jerry*	*John*
Anon.Session	*Granted*			
Tom.Credential		*Init*		
Tom.Id		*Init*	*Init*	*Init*
Tom.Name		*Init*		*Init*
Jerry.Credential			*Init*	
Jerry.Id			*Init*	*Init*
Jerry.Name			*Init*	*Init*
John.Credential				*Init*
John.Id			*Init*	*Init*
John.Name				*Init*

The goal used in this model is the "Specified goal" `AdminAction` discussed with the ones for vertical privilege escalation in Section 4.2.1 Page 70.

Attack trace(s)

Since the model is more complex than the one presented in the previous section, the `check` statement (and thus the model checking phase) requires an augment in the number of states, users and data:

```
check AdminAction for 4 State, 4 User, 21 Data
```

The resulting counterexamples and skolem constants are:

```
Trc0=[NoAction, Login, ListId, DeleteProfile]
Sko0=[3, TomProfile, John]
Trc1=[NoAction, Login, ListId, DeleteProfile]
Sko1=[3, JohnProfile, John]
Trc2=[NoAction, Login, ListId, DeleteProfile]
Sko2=[3, JerryProfile, John]
```

In the counterexamples, we can see how Trc_0, Trc_1 and Trc_2 are the same and refer to the actions that permit one to use the functionality `DeleteProfile` while in the three skolem constants the functionality is accessed by the administrative user `John` on the three available profiles.

Concretization parameters

Since the model checking phase deals with finding those execution fragments of the transition system that contain functionalities accessible only by administrative users, the concretization of such traces does not use any payload. In order to concretize the attack traces, the python code (i.e., the test execution engine) switch the information of a user admin with the ones of a normal user (i.e., not admin).

Results and comparison with other tools

The results of the concretization phase and the tests with other security tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	X	X	X

Even though the proposed framework was the only one that completed the WebGoat lesson, the fact that the other tools did not manage to find the attack confirms its complexity. I believe that for the security tools used as benchmark it is acceptable but is an added value for the proposed framework.

In this example, the success in finding the vulnerability relies in the fact that I have introduced the signature to differentiate the administrator in the model. With such signature I can reason about roles (as abstract security constraints) and derive traces that are interesting tests. I believe that the benchmark tools focus their analysis on raw data and do not reason about the meaning of the data they are handling.

Small conclusion

From a model checking perspective, the counterexamples are valid execution fragments of the real application (as stated before, the goal is to find those actions that are usable by admin users). The following switch between users' types (during the concretization phase) makes the test justifiable from a security perspective.

This example shows that it is possible to test weak access controls on the functionalities of a web application in a simple way. The main reason for this contribution resides in the model of the web application (that has been extended with a signature for the administrative role).

5.1.4 Role Based Access Control: Breaking Data Layer Access Control

Model and Goal

The model used in this lesson has the same structure of the one presented in Section 5.1.1. In order to make possible for a user to guess the values of the IDs of other users, the model has been extended with two actions, `ListIdGuessed` and `ViewProfileGuessed`, that can be executed by a user and have the same behavior of, respectively, `ListId` and `ViewProfile`. In order to maintain the knowledge of the users correctly handled by the model checker (i.e., without the possibility of data belonging to different types of knowledge), a transition definition for the possible guesses has been introduced:

```
fact { all s: State, s': s.next{
   ListIdGuessed in s'.action implies (
   s'.guessedK[s.user] = (s.guessedK[s.user] +
                          (Id - s.user.initialK - NoId )) &&
   (all u : User | (u != s.user) implies
        s'.guessedK[u] = s.guessedK[u])
   )}}
```

where for a user that performs a `ListIdGuessed`, the possible guesses are the IDs that are not part of the initial knowledge.

The initial state of the model is the same that I presented for the "Bypass Presentational Layer Access Control" in Section 5.1.3, and the goal is `GuessedId` in "Guessed data" as vertical privilege escalation in Section 4.2.1 Page 70.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check GuessedId for 4 State, 4 User, 21 Data
```

The resulting counterexamples are:

```
Trc0=[NoAction, Login, ListIdGuessed, ViewProfileGuessed]
Sko0=[3, JerryId, Tom]
Trc1=[NoAction, Login, ListIdGuessed, ViewProfileGuessed]
Sko1=[3, JohnId, Tom]
```

In the counterexamples, the attack traces are identical while the skolem constants refer to the user `Tom` that tries to guess the IDs of Jerry and John (not part of his initial knowledge).

Concretization parameters

As parameters for the test, I have used an array of possible IDs.

Results and comparison with other tools

The results of the concretization phase and the tests with other security tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	~	X

where

- The framework and Burp completed the lesson.
- With ZAP, I was unable to automatically replicate the attack (i.e., it was only possible by manually changing the request(s)).
- Paros did not find the attack.

Small conclusion

Within this model, I introduced the possibility for the users of guessing data. In this way I also augment the model checking capabilities of the framework with the possibility of extending the range of possible execution fragments that are derivable from the model. Regarding the concretization phase and the test, they are similar to brute forcing the valued of the IDs of the other users (the real values could not be known to the security analyst) but the use of such techniques is supported by the intended meaning of such data in the model.

5.1.5 AJAX Security: DOM-Injection

Model and Goal

The model used in this lesson is a “one action” model where the web application takes an activation key that allows the users to access a restricted functionality. The goal of the lesson is to enable the activate button and send a fake key to the web application.

In the Alloy model I have the possibility to insert the statement *disabled* in this action (i.e., the functionality) in order to deal with this problem during the test of the web application.

The initial state is the one where a guest user is interacting with the web application with a valid session:

	Guest.session
Guest	*Granted*

The goal used is `AJAXdisabled` shown in Section 4.2.6 Page 86.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check AJAXdisabled for 2 State, 1 User, 2 Data
```

The resulting counterexample is:

```
Trc0=[NoAction, SendLicenseKey]
Sko0=[1, GuestLicenseKey, Guest]
```

The counterexample contains the action that is disabled, and thus permits the test execution engine to perform tests for disabled functionalities.

Concretization parameters

From a programmatic perspective the functionality is disable in the input form:

```
<input disabled="" id="SUBMIT" value="Activate!"
      name="SUBMIT" type="SUBMIT">
```

Sending requests with python permits one to circumvent this limitation since the request is not sent via the user interface (i.e., a browser).

Results and comparison with other tools

The results of the tests with the security tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	X	X	X

The problem that the tools found with this type of test was the impossibility of using the functionality before the test. This made impossible for these tools to register the presence of the functionality, and thus using it during the tests.

Small conclusion

In this model, I give an example on how different typologies of functionalities can be modeled in our framework and how a security analyst can leverage information that other security tools can not see. Another possible extension of the tool leverages its modularity; in the preliminary version that I have implemented, the attack phase is hard coded, but a more complex implementation is possible. I can thus envision an automatic engine that can reason about the causes that are blocking the functionality.

5.1.6 AJAX Security: Dangerous Use of Eval

Model and Goal

In this lesson the security analyst is required to send some input containing a script and make it to reflect to her browser (which will execute the script).

The model contains two actions (i.e., the functionalities accessible in the page) even if they use the same parameters and have the same behavior. The initial state is again the one with the Guest user having a valid session:

	Guest.session
Guest	*Granted*

Since the page uses AJAX functionalities and the input is reflected on the page, in order to test this model I use the `AJAXxss` goal presented in Section 4.2.6 Page 86.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check AJAXxss for 2 State, 1 User, 8 Data
```

The resulting counterexamples are:

```
Trc0=[NoAction, UpdateCart]
Sko0=[1, GuestItem, Guest]
Trc1=[NoAction, Purchase]
Sko1=[1, GuestCC, Guest]
```

The counterexamples reveal that both the functionalities used in the model can be tested for XSS attacks.

Concretization parameters

In order to test the web application, I have introduced in the python engine a set of payloads that are specific for AJAX functionalities. As an example (and solution of the WebGoat lesson), the following payload is between those whose are been tested:

```
');alert(document.cookie);('
```

Results and comparison with other tools

The results of the tests performed with the framework and the security tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	X	✓	~	X

In this case, the outcome of the tests for each tool is

- WebGoat did not accept the payload sent by the framework even if correct (the problem resides the test execution engine encoding the payloads before sending them).
- Burp completed the lesson (with the same payload).
- ZAP found a reflected XSS but the WebGoat lesson was not reported as completed.
- Paros did not found the attack.

Small conclusion

In this model the outcome of the test has two main results. The test execution engine has been proved to be flawed while dealing with complex payloads (encoding the payloads could bring the web application to misinterpret them), on the other hand the model checking phase has been able of dealing with this typology of attacks.

5.1.7 Reflected XSS Attack

Model and Goal

The model used for this attack is the same as in the previous lesson (dangerous use of eval) with the difference that the web application does not implement the actions *update* and *purchase* through AJAX technologies.

Starting from the same initial state, the goal of this lesson is to find a reflected XSS attack (that can be expressed via the goal `XSS` presented in Section 4.2.3 Page 78).

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check XSS for 2 State, 1 User, 4 Data
```

The resulting counterexamples are:

```
Trc0=[NoAction, UpdateCart]
Sko0=[1, GuestItem, Guest]
Trc1=[NoAction, Purchase]
Sko1=[1, GuestCC, Guest]
```

As in the precious model, the skolem constants reveal that both the functionalities have to be tested for XSS.

Concretization parameters

The payloads used for this lesson are contained in a list of scripts; as an example, the following list contains some of the payloads used during the tests:

```
alert(String.fromCharCode(88,83,83,65,116,116,65,99,107))
<onmouseover="alert(1)"href="#">readthis!</a>
alert("XsSatt");
<script>alert(String.fromCharCode(88,83,83));</script>
red'onload='alert(1)'onmouseover='alert(2)
<script>alert("XSSatt");</script>
```

For every payload in the list, an expected values is saved in order to check if the attack can be triggered, e.g., the first payload should be executed and decoded as "*XSSAttAck*" by the browser.

Results and comparison with other tools

The results of the tests are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	✓	✓

All the tools solved the lesson without any problem.

Small conclusion

Having a simple lesson and a well known attack, the results of the tests were expected. Regarding the tests performed with the Burp intruder, all payloads returned the status code 200, it was impossible to see whether the payloads triggered the actual attack (it was possible after a manual revision requests made by the intruder).

5.1.8 XSS: Execute a Stored Cross-Site Scripting attack

Model and Goal

In this lesson, the security analyst is required to execute a Stored XSS attack against the *Street* field on the *EditProfile* page (as *Tom*) and verify that the user *Jerry* is affected by the attack.

The model of this lesson is the general model presented in Section 5.1.1. The initial state is defined as:

	A.ses	*T.cred*	*T.id*	*T.name*	*J.cred*	*J.id*	*J.name*
Anon	*Granted*						
Tom		*Init*	*Init*	*Init*			
Jerry			*Init*	*Init*	*Init*	*Init*	*Init*

Having to find a stored XSS the goal used is the `StoredXSS` for two agents (as required by the lesson) presented in Section 4.2.3 Page 78.

Attack trace(s)

Having two sessions, the number of states (i.e., the number of actions used during the attack) is higher than the previous cases:

```
check StoredXSS for 10 State, 2 User, 21 Data
```

and it is reflected on the counterexamples returned by the model checker:

```
Trc0=[NoAction, Login, ListId, ViewProfile, GetEdit,
      UpdateProfile, Logout, Login, ListId, ViewProfile]
Sko0=[5, TomProfile, 9, Tom, Jerry]
```

Concretization parameters

The concretization parameters used in this lesson are the same as the ones presented in the previous lesson (XSS). This is due to the fact that I am testing in the same attack but I am trying to trigger (i.e., provoke the script to run) it in a different way.

Results and comparison with other tools

Having to trigger the attack in a different location from the one where the payload is delivered, the results of the tests are positive only for the framework:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	X	X	X

Small conclusion

Even though the results of this lesson seems to elect the framework as the winner, I should add that the Burp Scanner can detect a stored XSS, with a full scan of the application, but this feature is not available in the free version used here.

In this example, the success of the framework resides in the attack traces that are generated by the model checker. The fact that the checking phase can be made in a different location from the one used during the attack phase makes the framework able of dealing with those vulnerabilities that require a complex control on the entry points of the attack and the pints where the attack has to be triggered.

5.1.9 XSS: Reflected XSS

Model and Goal

In this lesson, the general model presented in Section 5.1.1 is used along with the initial state presented in Section 5.1.8.

The goal of the lesson is to trigger a reflected XSS attack of the Search Staff page. In order to model check the model, I use the `XSScheck` goal for the checked data that is presented in Section 4.2.3 Page 78.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check XSScheck for 5 State, 3 User, 21 Data
```

The resulting counterexamples are:

```
Trc0=[NoAction, Login, ListId, GetSearch, Search]
Sko0=[4, TomName, Tom]
Trc1=[NoAction, Login, ListId, GetSearch, Search]
Sko1=[4, TomName, Jerry]
Trc2=[NoAction, Login, ListId, GetSearch, Search]
Sko2=[4, JerryName, Tom]
Trc3=[NoAction, Login, ListId, GetSearch, Search]
Sko3=[4, JerryName, Jerry]
```

Concretization parameters

The concretization of the attack is performed with the same payloads of the Stored XSS.

Results and comparison with other tools

The results of the tests performed with the tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	✓	X

From the tests, the success of the framework, Burp and ZAP emerged, while Paros failed to find the vulnerability.

Small conclusion

Even in this case, as for other simple vulnerabilities, the suitability of the framework is aligned with other tools even though for Burp all payloads returned the status code 200, making difficult to know for sure whether a payload could be triggered.

5.1.10 Command-Injection

Model and Goal

In this lesson, input data are used in an OS command that retrieve a file from the file system; the goal is to inject a command into the operating system.

The model is composed by one action and has for initial state:

	Guest.session
Guest	*Granted*

The goal used in this model is `FileAccessCommandInjection` presented in Section 4.2.5 as an instantiation of the general goal for Command-Injection on special functions.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check FileAccessCommandInjection for 2 State, 1 User, 2 Data
```

The resulting counterexample is:

$\texttt{Trc}_0$`=[NoAction, FileAccess]`
$\texttt{Sko}_0$`=[1, GuestFileAddress, Guest]`

Concretization parameters

The payloads used in order to test this vulnerability are contained in a list of OS commands that the test execution engine tries to fetch from the web application. As an example, the following list is a portion of the instantiation library used in this example:

```
" & netstat -an & ipconfig
" & ifconfig
|| ls
ls
```

Results and comparison with other tools

The results of the tests performed with the framework and the other tools were unsuccessful:

	Framework	Burp	OWASP ZAP	Paros
Result	X	X	X	X

Small conclusion

In this example, the attack was blocked by the security restriction in the server machine hosting WebGoat. This test has to be considered invalid even though the framework has been proven to been able to cope with this type of vulnerabilities. In order to test attacks that need special permissions on the target server an addition analysis is required.

5.1.11 Numeric SQL-Injection

Model and Goal

This model is composed by a single action as the web application displays a single form that allows a user to view some data. The goal of the lesson is to inject a SQL string that results in all the data being displayed.

As usual for the simple model the initial state is:

	Guest.session
Guest	*Granted*

The goal used in order to check the model is the `SQLInj` goal presented as a specification of the general goal in Section 4.2.4 Page 83.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check SQLInj for 2 State, 2 User,  5 Data
```

The resulting counterexamples are:

$\texttt{Trc}_0$`=[NoAction, SelectStation]`
$\texttt{Sko}_0$`=[1, GuestStation, Guest]`

Concretization parameters

As payloads for the tests, an array of general purpose SQL queries (or parts of them) has been used. As an example, the following list is part of the instantiation library used during the tests:

```
Administrator'--"
root'--
' HAVING 1=1 --"
1' or '1'='1"
1 or 1=1"
101 or 1=1!
```

Results and comparison with other tools

The results of the tests are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	X

The tests performed with the framework and Burp gave positive results, while the ones performed with ZAP and Paros did not manage to solve the lesson (i.e., find the vulnerability).

Small conclusion

In this scenario the framework is aligned with the Burp suite; 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. The success of the attack is due to the payloads used during the tests. It is important to mention that tools like Paros are not updated too often and the introduction of new payloads is thus difficult. With the instantiation library used in the framework the task of introducing new payloads (or vulnerabilities) as simple as copying them into a textual file.

5.1.12 Log Spoofing

Model and Goal

This lesson displays two ares: (i) a login form and (ii) an area that represents what is going to be logged in the web server's log file. The goal is to make the web application believe that a username "admin" has succeeded in logging in and add a script to the log file.

The model is composed by a single login action, and has as initial state:

	Guest.session
Guest	*Granted*

The goal used in the model is `SQLInjFileSystem` presented in Section 4.2.4 Page 83.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check SQLInjFileSystem for 2 State, 1 User,  2 Data
```

The resulting counterexample is:

```
Trc0=[NoAction, Login]
Sko0=[1, GuestCredential, Guest]
```

Concretization parameters

The payloads used in this lesson are a combination of a log spoofing attack:

```
Smith%0d%0aLogin Succeeded for username: admin
```

and a XSS attack:

```
<script>alert(document.cookie)</script>
```

Both payloads are required in order to complete the lesson.

Results and comparison with other tools

All tools were able of solving the lesson.

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	✓	✓

Small conclusion

The web application implemented in this lesson, even though very simple, takes into consideration a vulnerability that is increasingly common in web applications since the login functionality often lacks the proper input sanitization. The fact that ad-hoc payloads can be delivered with the framework (and the related attack discovered) increases the strength of the testing capabilities of the framework itself.

5.1.13 XPATH-Injection

Model and Goal

In this model (composed of a single action) the security analyst can test another vulnerability that can affect the login functionality of a web application. The login functionality is used in order to display some data belonging to a user that delivers correct credentials. The lesson's goal is to display also the data of other employees..

The initial state of the model is defined as:

	Guest.session
Guest	*Granted*

Regarding the goal, in this model I have used the `SQLInj` goal (the correct selection of the payloads related to this goal is made during the concretization phase).

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check SQLInj for 2 State, 1 User,  6 Data
```

The resulting counterexample is:

```
Trc0=[NoAction, Login]
Sko0=[1, GuestCredential, Guest]
```

Concretization parameters

The payloads used in this lesson are the ones for the SQL-Injection.

Results and comparison with other tools

The results of the tests performed with the framework and the security tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	X

where only the framework and Burp were able to solve the lesson.

Small conclusion

With this lesson, I have shown another vulnerability of the login functionality. As I will show in the string SQL-Injection and brute force examples a security analyst can check different attacks against the same functionality (e.g., Login). In order to do so, a security analyst has to change the goal that is checked during the model checking phase. The tests of the resulting counterexamples are thus performed for the attacks the goals refer to. This feature can be leveraged when a thorough analysis of a web application has to be performed.

5.1.14 String SQL-Injection

Model and Goal

In this lesson, the web application is implemented as a single form that allows users to view their credit card numbers. The goal is to inject a SQL string that results in all the credit card numbers being displayed. As for the other types of SQL-Injection, the model starts in a state where the only user of the system already has a valid session (the goal remain `SQLInj`).

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check SQLInj for 2 State, 1 User,  6 Data
```

The resulting counterexample is:

```
Trc0=[NoAction, Search]
Sko0=[1, GuestName, Guest]
```

Concretization parameters

The payloads used in this lesson are the ones for the SQL-Injection.

Results and comparison with other tools

The results of the tests performed with the framework and the other tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	X

Small conclusion

With this lesson, I show another vulnerability that could affect the web applications that use a non-sanitized SQL query. The framework has been proven successful in the test and aligned to the burp suit. I believe that the results are due to the fact that the payloads used during the tests performed with the framework and Burp were the same. This remarks the importance of having the possibility of expanding the instantiation library of the framework.

5.1.15 SQL-Injection: String SQL-Injection

Model and Goal

The model used in this lesson is the one presented as a general model for WebGoat in Section 5.1.1. The initial state of the model has been defined as:

	A.ses	*T.cred*	*T.id*	*T.name*	*J.cred*	*J.id*	*J.name*
Anon	*Granted*						
Tom		*Init*	*Init*	*Init*		*Init*	*Init*
Jerry			*Init*	*Init*	*Init*	*Init*	*Init*

Since the goal of the lesson is to use a String SQL-Injection to bypass the authentication mechanism the goal used in the model is `BypassLoginSQL` presented in Section 4.2.4 Page 83.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check BypassLoginSQL for 2 State, 2 User,  6 Data
```

The resulting counterexamples are:

$\texttt{Trc}_0$=[NoAction, Login]
$\texttt{Sko}_0$=[1, Jerry]
$\texttt{Trc}_1$=[NoAction, Login]
$\texttt{Sko}_1$=[1, Tom]

Concretization parameters

The payloads used in this lesson are the ones for the SQL-Injection.

Results and comparison with other tools

The results of the tests performed with the security tools are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	X

Small conclusion

In this lesson, I have presented how the framework can use String SQL-Injection to bypass the authentication mechanism in order to gain access to an account without knowing the password of the target user. The versatility of the framework permits to test the login functionality for a variety of attacks (depending on the goal selected during the model checking phase).

5.1.16 Insecure Configuration: Forced Browsing

Model and Goal

In this lesson, the goal is to guess the URL for the “config” interface that is only available to the maintenance personnel (the application does not check for horizontal privileges).

The model is composed by a single action (`FileInclusion`), and its initial state is the one where a user of the system has a valid session. The goal used in this lesson is `FileInclusion` (as the only action available in the model). In order to test this type of vulnerabilities, I use the location of the action as a base for launching a forced browsing attack (see Section 4.2.1 Page 73 for further details on the goal).

Attack trace(s)

Since the model is composed by a single action, the `check` statement and the resulting counterexample are again minimal in the number of states:

```
check FileInclusion for 2 State, 1 User, 1 Data

       Trc0=[NoAction, FileInclusion]
       Sko0=[1, GuestFileAddress, Guest]
```

Concretization parameters

As an example, the following payloads are part of the instantiation library used during the tests:

```
.htpasswd
.meta
.web
../../../../../../etc/passwd
apache/logs/access.log
```

Results and comparison with other tools

The results of the tests are depicted in the following table:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	~	X

where

- The framework and Burp were able to solve the lesson.
- With ZAP the lesson was not completed (even though ZAP is able of performing the attack the successful payload was not available in the tested lists).
- Paros was not able to solve the lesson.

Small conclusion

In this example, the framework has been able of performing a forced browsing attack from a given location against the web application. Since this type of attack mainly depends on the instantiation library (i.e., the set of locations to be tested), the completion of the lesson only gives a valid indicator of the successfulness of the payloads used and the possibility of performing this type of tests.

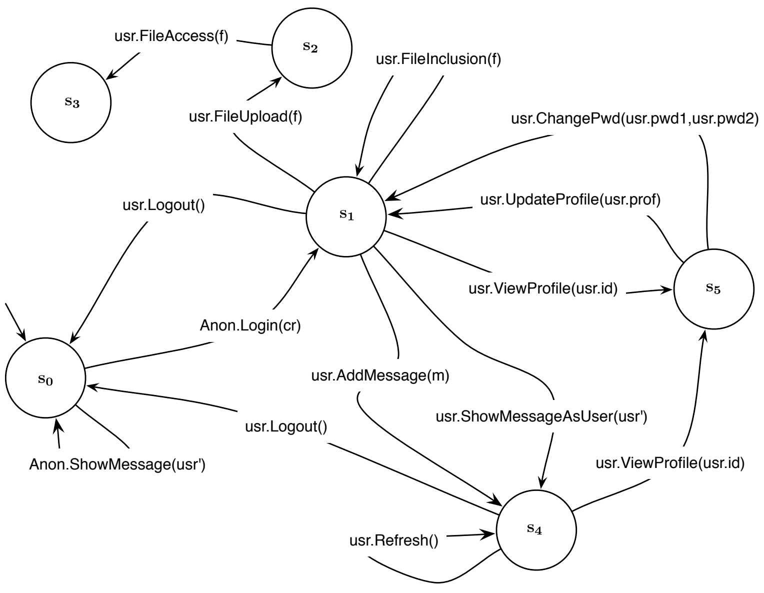


Figure 5.2: Gruyere high-level model.

5.2 Gruyere

In this section, I show the definition of the model, the results of the model checking phase and tests for the Gruyere case study.

Gruyere [13] is a small web application that allows its users to publish snippets of text and store assorted files. Gruyere has multiple security bugs ranging from cross-site scripting and cross-site request forgery, to information disclosure, denial of service, and remote code execution. The goal of this codelab is to guide the user through discovering some of these bugs and learning ways to fix them both in Gruyere and in general.

Gruyere is written in Python, however, the security vulnerabilities covered are not Python-specific and the user can do most of the lab without even looking at the code.

5.2.1 Model

The Gruyere model (depicted in Figure 5.2) is composed by actions that permit users to

- add/delete messages (snippets),
- upload files,
- modify their profile, and
- see other user profiles.

The definition of the actions is reported in Table 5.2.

The data used in the model are the ones in the following tuple:

```
UserData =
    (
    credential,
    id,
    profile,
    fileAddress,
    password1,
    password2,
    text,
    session,
    )
```

The data used in the model depict a system where each user has credentials, one identifier, a profile, two distinct passwords (used in order to change his password), some text he can use to write messages, and a possible session with the web application (granted through the login phase).

For this model, I have defined three distinct users (the *Anon* user and two agents *Usr1* and *Usr2*) and defined the initial state of the transition system as the one where none of the users have a valid session with the web application:

	Guest	*Usr1*	*Usr2*
Guest.ses	*Granted*		
Usr1.Pwd1		*Init*	
Usr1.Pwd2		*Init*	
Usr1.Cred		*Init*	
Usr1.Text		*Init*	
Usr2.Pwd1			*Init*
Usr2.Pwd2			*Init*
Usr2.Cred			*Init*
Usr2.Text			*Init*

The DVWA web application, as the target of the tests, has been modeled in the general model (that includes the initial state). This model is used in all the scenarios that I discuss in the following sections.

5.2.2 File Upload XSS

Goal

The objective of this attack is to upload a file that allows one to execute an arbitrary script in the web application. In order to test this type of vulnerability, I use the `XSS_FileUpload` goal presented in Section 4.2.3 Page 78 as a specified goal for XSS attacks.

Table 5.2: Definition of the actions for the Gruyere case study.

```
$Login(x,\ x.credential)$
 if $M[Anon, Anon.session]$ = $Granted$
 if $M[x, x.cred]$ = $Initial$
  Reset $M$ for $x$
  Del $Granted$ into $M[Anon, Anon.session]$
  Add $Granted$ into $M[x, x.session]$
  Add $Checked$ into $M[x, x.credential]$
End

$Logout(x)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
  Reset $M$ for $x$
  Del $Granted$ into $M[x, x.session]$
  Add $Granted$ into $M[Anon, Anon.session]$
End

$ChangePwd(x)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
 if $M[x, x.profile]$ = $Edit$
 if $M[x, x.Pwd1]$ = $Initial$
  Reset $M$ for $x$
  Add $Checked$ into $M[x, x.Pwd1]$
  Add $WriteDB$ into $M[x, x.Pwd2]$
End

$ShowMessage(x, y)$
 if $M[x, x.session]$ = $Granted$
 if $y$ != $Anon$
  Reset $M$ for $x$
  Add $ShowDB$ into $M[x, y.text]$
End

$FileUpload(x)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
  Reset $M$ for $x$
  Add $WriteFS$ into $M[x, x.fileAddr]$
End

$Refresh(x)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
 if $M[x, x.text]$ = $showDB$
  Reset $M$ for $x$
  Add $ShowDB$ into $M[x, x.text]$
  Add $AJAX$ into $M[x, x.text]$
  Add $echo$ into $M[x, x.text]$
End

$ViewProfile(x)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
  Reset $M$ for $x$
  Add $ShowDB$ into $M[x, x.profile]$
  Add $Edit$ into $M[x, x.profile]$
End

$UpdateProfile(x,\ x.prof)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
 if $M[x, x.profile]$ = $Edit$
  Reset $M$ for $x$
  Add $WriteDB$ into $M[x, x.profile]$
End

$AddMessage(x, x.text)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
  Reset $M$ for $x$
  Add $WriteDB$ into $M[x, x.text]$
End

$ShowMessageAsUser(x, y)$
 if $M[x, x.session]$ = $Granted$
 if $y$ != $Anon$
 if $x$ != $Anon$
  Reset $M$ for $x$
  Add $ShowDB$ into $M[x, y.text]$
End

$FileAccess(x)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
  Reset $M$ for $x$
  Add $ShowFS$ into $M[x, x.fileAddr]$
End

$FileInclusion(x)$
 if $M[x, x.session]$ = $Granted$
 if $x$ != $Anon$
  Reset $M$ for $x$
  Add $PageIncluded$ into $M[x, x.fileAddr]$
  Add $ShowFS$ into $M[x, x.fileAddr]$
End
```

Attack trace(s)

During the model checking phase the following `check` has been used: statement:

```
check FileUpload for 3 State, 3 User, 21 Data
```

The resulting counterexamples are:

$\texttt{Trc}_0$`=[NoAction, Login, FileUpload, FileAccess]`
$\texttt{Sko}_0$`=[2, Usr2FileAddress, 3, Usr2]`
$\texttt{Trc}_1$`=[NoAction, Login, FileUpload, FileAccess]`
$\texttt{Sko}_1$`=[2, Usr1FileAddress, 3, Usr1]`

Concretization parameters

The exploitation of this vulnerability is made possible by the fact that an attacker can upload HTML files and these files can contain scripts. In order to concretize the attack, I have created an HTML file containing the script:

```
<script>
   alert(document.cookie);
</script>
```

and submitted it (via the python engine) to the web application.

Results and comparison with other tools

The results of the tests performed with the security tools and the framework are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	NA	NA	NA

In this scenario, the proposed framework was able to test the vulnerability and check the presence of the attack; for the other tools, even though they were not able to find the attack, I do not give a negative response because this type of attack is usually tested manually and thus results out of scope.

Small conclusion

With this example, I show how it is possible to automatize the research of not trivial attacks through the framework using all the phases that compose it:

- the model checking phase in order to find the entry points,
- the concretization phase in order to define the payloads, and
- the python engine to check the presence of the actual attack.

5.2.3 Reflected XSS

Goal

Gruyere suffers from a possible reflected XSS attack launched via a URL. In order to check this vulnerability, I use the `urlXSS` goal that I have defined as a specification of possible XSS attacks in Section 4.2.3 Page 78.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check urlXSS for 3 State, 1 User, 1 Data
```

The resulting counterexamples are:

```
Trc0=[NoAction, Login, FileInclusion]
Sko0=[2, Usr1FileAddress, Usr1]
Trc1=[NoAction, Login, FileInclusion]
Sko1=[2, Usr2FileAddress, Usr2]
```

Concretization parameters

For this example the payloads for XSS presented in the previous sections have been used.

Results and comparison with other tools

The results of the tests are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	X	X	~

where:

- The framework was able to test and confirm the attack.
- Burp delivered all the payloads but it was not possible to trigger the attack.
- ZAP did not find the attack.
- Paros did find an attack but on a different location of the web application.

```
deletesnippet?index=
/snippets.gtl?uid=
```

Small conclusion

In this scenario, the location of the entry point for the attack is a major problem for the benchmark tools. The framework was able to find and confirm the attack primarily for the used 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 research of many vulnerabilities.

5.2.4 Stored XSS

Goal

The goal used in this example is `StoredXSS` for one agent presented in Section 4.2.3 Page 78.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check StoredXSS for 4 State, 2 User, 21 Data
```

The resulting counterexamples are:

$\texttt{Trc}_0$`=[NoAction, Login, AddMessage, ShowMessageAsUser]`
$\texttt{Sko}_0$`=[2, Usr1Text, 3, Usr1, Usr1]`
$\texttt{Trc}_1$`=[NoAction, Login, AddMessage, ShowMessageAsUser]`
$\texttt{Sko}_1$`=[2, Usr2Text, 3, Usr2, Usr2]`

Concretization parameters

For this attack, I use the payloads for XSS presented in Section 5.1.7 Page 116.

Results and comparison with other tools

	Framework	Burp	OWASP ZAP	Paros
Result	✓	~	X	X

In this case, the Burp intruder was able to deliver the payloads but I had to check manually if it was possible to trigger the attack. As before, during the tests I used the Burp intruder not the Burp analyzer (which is able of analyzing multiple locations and thus to find stored XSS attacks).

Small conclusion

When the attack is triggered in a different location from the one used as entry point the tested tools did not find the attack while the framework is able to perform and complete the test.

5.2.5 Stored XSS via HTML Attribute

Goal

In this example, the aim of the attack is to inject a value in an HTML attribute of a profile. For this attack, the goal for stored XSS has been rewritten in order to target the search of counterexamples (and thus the testing phase) to a subset of data (`StoredXSSaimed` in Section 4.2.3 Page 78).

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check StoredXSSaimed for 5 State, 2 User, 21 Data
```

The resulting counterexamples are:

Trc_0=[NoAction, Login, ViewProfile, UpdateProfile, ViewProfile]
Sko_0=[3, Usr1Profile, 4, Usr1, Usr1]
Trc_1=[NoAction, Login, ViewProfile, UpdateProfile, ViewProfile]
Sko_1=[3, Usr2Profile, 4, Usr2, Usr2]

Concretization parameters

For this attack, I use the payloads for XSS presented in Section 5.1.7 Page 116.

Results and comparison with other tools

The comparison in finding the vulnerability for the framework and the security tools is:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	X

In this scenario, the framework and Burp were able to find the vulnerability, while ZAP and Paros were not. A special note has to be made for Paros that reports a possible SQL-Injection

```
saveprofile?action=new&uid=1%20AND%201=2&is_author=True
```

even if Gruyere does not use a database (the server is run via python). I believe that Paros mistake relies in the facts that it tests web applications with general payloads and methodologies, and probably it assumes the technologies used from the locations of the tests.

Small conclusion

In this example, I have shown how the framework can be used in order to target specific areas of a web application. In the case a security analyst decides to use the general goal for stored XSS (`StoredXSS`) (along with the same `check` statement used in this example), the framework tests both the counterexamples presented in the previous section (Stored XSS) and the ones presented here.

5.2.6 Reflected XSS via AJAX

Goal

The aim of this example is to find an XSS attack that uses a bug in Gruyere's AJAX code. The attack should be triggered when the user clicks the refresh link on the page (modeled as an action).

In this scenario, the goal used is `AJAXxss` presented in Section 4.2.6 Page 86.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check AJAXxss for 4 State, 1 User, 21 Data
```

The resulting counterexamples are:

```
Trc0=[NoAction, Login, ShowMessageAsUser, Refresh]
Sko0=[3, Usr1Text, Usr1]
Trc1=[NoAction, Login, ShowMessageAsUser, Refresh]
Sko1=[3, Usr2Text, Usr2]
```

Concretization parameters

For this attack, I use the payloads for XSS along with some payloads for AJAX technologies.

Results and comparison with other tools

	Framework	Burp	OWASP ZAP	Paros
Result	✓	X	X	X

Among the tested tools, the framework was the only one able to find the attack.

Small conclusion

In this case, the location and the procedure used to deliver the payload and trigger the attack make the framework successful in finding the attack.

5.2.7 Information Disclosure via Path Traversal

Goal

In this example, Gruyere asks to find a way to read a file (“secret.txt”) from the server.

As before, the attack used is `FileInclusion` because I use the location of the action as a base for launching the path traversal attack (see Section 4.2.1 Page 73 for further details on the goal).

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check FileInclusion for 3 State, 1 User, 1 Data
```

The resulting counterexamples are:

```
Trc0=[NoAction, Login, FileInclusion]
Sko0=[2, Usr1FileAddress, Usr1]
Trc1=[NoAction, Login, FileInclusion]
Sko1=[2, Usr2FileAddress, Usr2]
```

The counterexamples returned by the Alloy analyzer state that it is possible to launch the attack after a login, and the attack trace (since $Trc_0 = Trc_1$) has to be tested for both the available users.

Concretization parameters

For this attack, I use the payloads for “force browsing” presented in the previous sections (and modified for path traversal) with the addition of the payload `..%2fsecret.txt` (required for the example).

Results and comparison with other tools

	Framework	Burp	OWASP ZAP	Paros
Result	✓	X	~	~

The results of the tests performed with the tools are not unexpected since the payload to be used is too specific to be available in the tools’ lists of directory/files. The only problem arises with the Burp intruder, which failed to find the attack even if the proper name of the file was given (this behavior is due to an encoding of the URL that Gruyere failed to interpret correctly).

Small conclusion

As for the force browsing lesson in WebGoat, also in this example, the framework has been able of performing a forced browsing attack from a given location against the web application.

5.3 Damn Vulnerable Web Application

In this section, I show the definition of the model, the results of the model checking phase and tests for the Damn Vulnerable Web Application (DVWA) case study.

Damn Vulnerable Web Application (DVWA) [62] is a PHP/MySQL web application that suffers for various vulnerabilities. Its main goals are to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and aid teachers/students to teach/learn web application security in a class room environment.

5.3.1 Model

The general model of DVWA (depicted in Figure 5.3) shows that the application is developed as a set of pages. Every page contains a vulnerable entry point (i.e., one of the functionalities available to the users) accessible by the security analyst. The model depicted in Figure 5.3 is a simplified version of the model used in this example since the state s_0 is reachable through a login phase on the web application (used to log in the user that performs the attacks); the first login phase is deliberately not attackable. A `NoAttack` statement has been introduced in the model in order to skip the first login action during the research of counterexamples. This is due to the fact that another login functionality is implemented in the web application and the first one is not meant to be attacked.

The definition of the actions in Figure 5.3 is reported in Table 5.3.

The data used by the actions of DVWA are the ones in the following tuple:

```
UserData =
    (
    credential,
    id,
    profile,
    fileAddress,
    password1,
    password2,
    text,
    ip,
    message,
    session,
    )
```

Since the functionalities are "stand alone" (i.e., in the model every single data is used only by one functionality) and one single user is required to attack the application. The initial state for the transition system that I use is the one where a `Guest` user has a valid session with the web application

Table 5.3: Definition of the actions for the DVWA case study.

```
Login(x, x.credential)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, x.cred] = Initial
  Reset M for x
  Add Granted into M[x, x.session]
  Add Checked into M[x, x.credential]
End
```

```
Logout(x)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Del Granted into M[x, x.session]
  Add Granted into M[Anon, Anon.session]
End
```

```
ChangePwd(x, x.pwd1, x.pwd2)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, x.pwd2] = Initial
  Reset M for x
  Add Checked into M[x, x.pwd1]
  Add WriteDB into M[x, x.pwd2]
End
```

```
InputEcho(x, x.text)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add Echo into M[x, x.text]
End
```

```
SendMessage(x, x.message)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add WriteDB into M[x, x.message]
  Add ShowDB into M[x, x.message]
End
```

```
FileAccess(x, x.fileAddr)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add showFS into M[x, x.fileAddr]
End
```

```
LoginNoAttack(x, x.credential)
 if M[Anon, Anon.session] = Granted
 if M[x, x.cred] = Initial
  Reset M for x
  Del Granted into M[Anon, Anon.session]
  Add Granted into M[x, x.session]
  Add Checked into M[x, x.credential]
  Add NoAttack into M[x, x.credential]
End
```

```
SearchForm(x, y.id)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add Checked into M[x, x.id]
  Add ShowDB into M[x, y.profile]
End
```

```
PingBox(x, x.ip)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add Exec into M[x, x.ip]
  Add ShowSD into M[x, x.message]
End
```

```
FileUpload(x, x.fileAddr)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add WriteFS into M[x, x.fileAddr]
End
```

```
FileInclusion(x, x.fileAddr)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add PageIncluded into M[x, x.fileAddr]
End
```

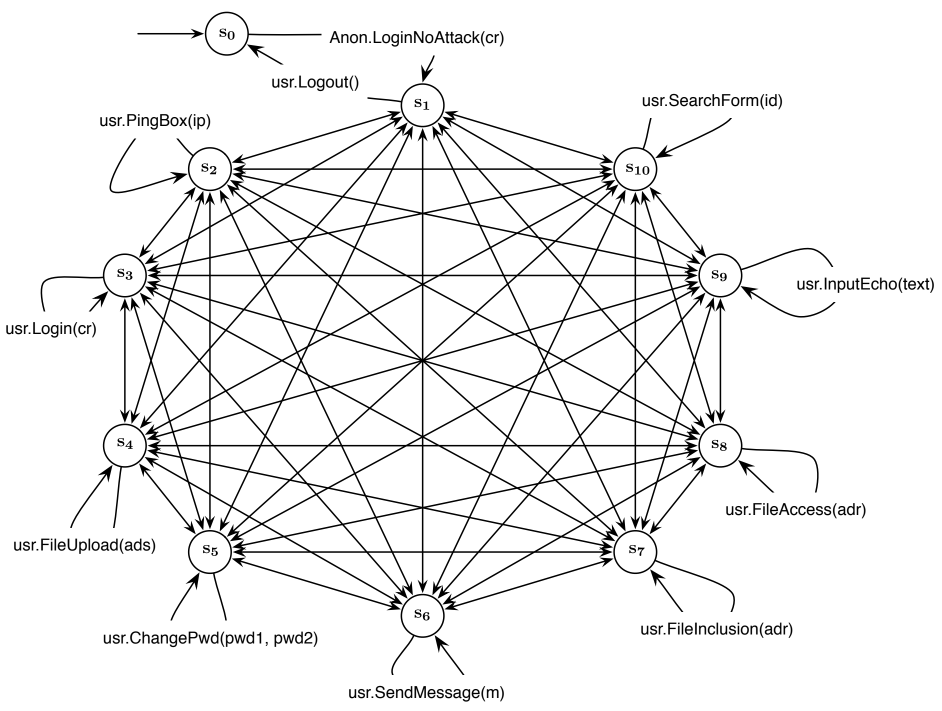


Figure 5.3: DVWA high-level model.

and knows some credentials (that will be used in order to test a second login phase) and a password:

	Anon	*Admin*
Guest.ses	*Granted*	
Adm.Cred		*Init*
Adm.Pwd1		*Init*

5.3.2 Brute Force

Goal

In this example, the application displays a login form that the security analyst can test through brute-forcing techniques.

The goal used in order to test this type of attack is `PasswordBruteForce` (described in Section 4.2.7 Page 86).

Attack trace(s)

The `check` statement used in this model and the resulting counterexample reflect the simplicity of the model (i.e., the small number of states required to find a counterexample):

```
check PasswordBruteForce for 3 State, 2 User, 15 Data
```

```
Trc0=[NoAction, LoginNoAttack, Login]
Sko0=[2, AdminCredential, Admin]
```

Concretization parameters

In order to test this attack, I use a list of passwords, e.g.,

```
admin
test
john
12345
qwerty
qwertz
asd
administrator
watson
```

and the message

```
Welcome to the password protected area
```

as an indicator for the success of the attack.

Results and comparison with other tools

The results of the tests are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	~	*NA*

The results have to be read as follows:

- The framework and Burp completed the task successfully (with Burp a manual check of the results was needed).
- With ZAP the attack is possible with some work on the payloads.
- Paros does not permit security to the analyst to perform brute force attacks.

Small conclusion

In this example, I show how the framework can be used in order to automatize brute forcing attacks. The possibility of defining how an attack has to be checked (i.e., if it is successful or not) makes the framework able of bringing the automatization a step further and lowering the discovery of false positive attacks (even if a good knowledge of the web application is required for the security analyst).

5.3.3 Command Execution

Goal

In this part of the web application, a form permits the user to ping an IP address. The attack aims at executing a command on the target server. In order to test this vulnerability, the goal used in the model is `CommandExec` presented in Section 4.2.5 Page 85.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check CommandExec  for 3 State, 2 User, 21 Data
```

The resulting counterexample are:

```
Trc0=[NoAction, LoginNoAttack, PingBox]
Sko0=[2, AdminIp, Admin]
```

Concretization parameters

The payloads used in this example are the ones for Command-Injection presented in Section 5.1.10.

Results and comparison with other tools

The results of the tests are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	X

In this scenario only the framework and the Burp intruder were able of successfully completing the tests (i.e., find the vulnerability).

Small conclusion

As already discussed for the WebGoat lesson in Section 5.1.10, the payloads used during the tests of this type of attack are one of the key factors in discovering the vulnerability. The capability of the proposed framework to perform this type of analysis is confirmed.

5.3.4 File Inclusion

Goal

In this example, the displayed page is loaded through a ‘`page` parameter (i.e., `?page=index.php`), the goal is to determine which files are included.

The goal used in the model is `FileInclusion` (Section 4.2.1 Page 73 and as already used in Section 5.1.16).

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check FileInclusion for 3 State, 2 User, 1 Data
```

The resulting counterexample is:

```
Trc0=[NoAction, LoginNoAttack, FileInclusion]
Sko0=[2, AdminFileAddress, Admin]
```

Concretization parameters

The payloads used are the same as the forced browsing attack in Section 5.1.16 for WebGoat (i.e, a list of resources to be tested).

Results and comparison with other tools

The test results show the capability of the framework (along with Burp) of finding the vulnerability:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	X

Small conclusion

The framework has been able of performing the attack. This only gives a valid indicator on the possibility of performing this type of tests since they depend on the set of payloads used rather than on the testing methodologies.

5.3.5 SQL-Injection and Blind SQL-Injection

Goal

The goal used in this example is `SQLInj` (see Section 4.2.4 Page 83).

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check SQLInj for 3 State, 3 User, 21 Data
```

The resulting counterexample are:

```
Trc0=[NoAction, LoginNoAttack, SearchForm]
Sko0=[2, AdminId, Admin]

Trc1=[NoAction, LoginNoAttack, SearchForm]
Sko1=[2, AdminId, Admin]
```

The functionalities used for the two attacks are the same. In the blind SQL-Injection (*Trc1* and *Sko1*) the error messages from the database are filtered and thus also the counterexamples of the two attacks are the same.

Concretization parameters

The payloads used are the ones for SQL-Injection presented in the previous sections for this type of tests.

Results and comparison with other tools

The results of the tests are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	✓
Result	✓	✓	X	✓

Small conclusion

The results of the tests are not unexpected (apart from the unsuccessful test with ZAP). SQL-Injection vulnerabilities are well known and the testing methodologies of the tools are mature. In this example, the framework is aligned with the benchmark tools.

5.3.6 File Upload

Goal

The goal used for this example is `XSS_FileUpload` defined as a specification of XSS attack in Section 4.2.3 Page 78.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check FileUpload for 4 State, 3 User, 21 Data
```

The resulting counterexample are:

$\texttt{Trc}_0$`=[NoAction, LoginNoAttack, FileUpload, 'FileAccess']`
$\texttt{Sko}_0$`=['2', 'AdminFileAddress', '3', 'Admin']`

Concretization parameters

The payload used is the one presented in Section 5.2.2.

Results and comparison with other tools

As in the example presented in Section 5.2.2, the proposed framework was able to test the vulnerability and check the presence of the attack:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	NA	NA	NA

Small conclusion

With this example, I show how it is possible to automatize the research of not trivial attacks. For the benchmark tools, I do not give a negative evaluation because this type of attack is for them out of scope. The possibility of performing this type of tests with the framework is an added value for the security analysts that can automatize the tests without performing them manually (with a considerable investment of time).

5.3.7 Reflected XSS

Goal

In this example, the page contains a form that takes an input and returns a message containing the input data. The goal used in the model is `XSS` presented in Section 4.2.3 Page 78.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check XSS for 3 State, 3 User, 21 Data
```

The resulting counterexample is:

```
Trc0=[NoAction, LoginNoAttack, InputEcho]
Sko0=[2, AdminText, Admin]
```

Concretization parameters

The payloads used for this example are the ones for XSS presented in Section 5.1.7.

Results and comparison with other tools

The results of the tests are:

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	✓

where only ZAP was not able to find the vulnerability.

Small conclusion

In this example, the suitability of the framework in performing a reflected XSS attack is aligned with the benchmark tools. As for other examples, this attack is well known and modern security tools are using mature testing methodologies. In this scenario, the framework can be used as an alternative to such tools.

5.3.8 Stored XSS

Goal

In this example, the target functionality is a form that the user can use to leave a message on a guestbook. The goal used is `StoredXSS` presented in Section 4.2.3 Page 78.

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check StoredXSS for 4 State, 3 User, 21 Data
```

The resulting counterexample are:

```
Trc0=[NoAction, LoginNoAttack, SendMessage, SendMessage]
Sko0=[2, AdminMessage, 3, Admin, Admin]
```

Concretization parameters

The payloads used for this example are the ones for XSS presented in Section 5.1.7.

Results and comparison with other tools

	Framework	Burp	OWASP ZAP	Paros
Result	✓	✓	X	✓

Also in this case, only ZAP was not able to find the vulnerability.

Small conclusion

As I presented in the previous instances of stored XSS, the framework has been able of finding the vulnerability but, in this case, two of the tested tools were also able to find the vulnerability. This is due to the fact that the messages delivered through the functionality are directly displayed as the result of the submission and thus the tools are able to directly check the attack.

5.4 OnlineShop

In this section, I discuss the model and the model checking results for the OnlineShop case study.

The OnlineShop is a fictitious case studies where I assume to be modeling a web application for electronic commerce. This made up application has the features that most online shops have, i.e.,

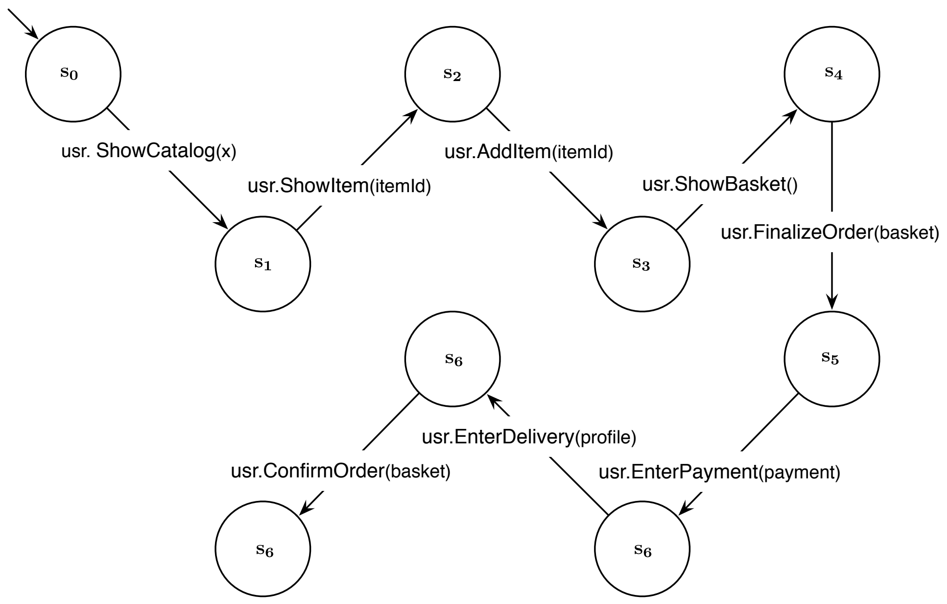


Figure 5.4: OnlineShop high-level model.

- a catalog of products to be sold, and
- a basket for each user where items can be added before the purchase.

For this web application, I assume that, in order to purchase some goods, a user has to (i) finalize the order (i.e., confirming the will of purchasing the items contained in the basket), (ii) enter the payment information and the delivery information, and in the end (iii) confirm the order (i.e., the information that he gives during this process).

This section does not contain any result for the tests because the model has been defined for a generic web application and thus the concretization of counterexamples (from the model checking phase) has not been tested on any real web application.

5.4.1 Model

The general model of OnlineShop (depicted in Figure 5.4) refers to the expected workflow that a user has to follow in order to complete a transaction with the web application (i.e., purchase some items). The actions contained in the model (the definition of which is given in Appendix 5.4) refer to the functionalities of a web application where a user can browse a catalog of items, view an item, add an item to his basket, send delivery and payment information, and conclude the order.

The data structure used in this model is:

Table 5.4: Definition of the actions for the OnlineShop case study.

```
ShowCatalog(x)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add ShowBD into M[x, x.catalog]
End
```

```
ShowItem(x, itemId)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, x.catalog] = showDB
  Reset M for x
  Add ShowSD into M[x, x.item]
End
```

```
AddItem(x, itemId)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, x.item] = showSD
  Reset M for x
  Add WriteSD into M[x, x.basket]
End
```

```
ShowBasket(x)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add ShowSD into M[x, x.basket]
End
```

```
FinalizeOrder(x, x.basket)
 if M[x, x.session] = Granted
 if x != Anon
 if M[x, x.basket] = showSD
  Reset M for x
  Add Checked into M[x, x.basket]
End
```

```
EnterPayment(x, payment)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add WriteDB into M[x, x.payment]
  Add Checked into M[x, x.payment]
End
```

```
EnterDelivery(x, x.profile)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add WriteDB into M[x, x.profile]
  Add Checked into M[x, x.profile]
End
```

```
ConfirmOrder(x, x.basket)
 if M[x, x.session] = Granted
 if x != Anon
  Reset M for x
  Add WriteDB into M[x, x.basket]
  Add Checked into M[x, x.basket]
End
```

UserData =
(
catalog = (itemId, ...),
userId,
profile = (name, surname, address),
payment = (bankAccount,...),
basket = (itemId, ...),
item = (itemId, description, name,...),
session,
)

The initial state of the transition system that I use is the one where a user has a valid session with the web application:

	Anon	*Admin*
Usr.ses	*Granted*	

In this case study, I do not model multiple users and thus the model checking phase is carried out with a single user that already has a valid session with the web application. In the following, I show the results of the model checking phase with respect to the logic flaws discussed in Section 4.2.2 Page 75.

5.4.2 Check Payment

Goal

One of the security requirements for a web application like the one I am modeling is that it is not possible to confirm an order without paying. The Alloy goal used in order to test this flaw is `CheckPayment` (discussed in Section 4.2.2 Page 77).

Attack trace(s)

During the model checking phase the following `check` has been used:

```
check CheckPayment for 8 State, 1 User, 20 Data
```

The resulting counterexample is:

$\texttt{Trc}_0$`=[NoAction, ShowCatalog, ShowItem, AddItem, ShowBasket,`
`FinalizeOrder, EnterDelivery, ConfirmOrder]`
$\texttt{Sko}_0$`=[7, UserPayment]`

Small conclusion

This example shows how the framework can be used in order to test logic vulnerabilities where important checks are skipped by the web application. The definition of the goal reflects the understanding of the security analyst of the process workflow (implemented by the web application) and thus represents the actual flaw (i.e., it contains the assumption on the workflow that the flaw exploits). The benchmark tools used in the previous sections focus their tests on well known vulnerabilities and lack in understanding the "logic" behind a tested web application. With this example I show that the model checking phase of the framework can be used in order to derive possible tests for logic flaws (I believe, with promising results).

5.4.3 Skip Stages

Model and Goal

As discussed in Section 4.2.2, one of the possible tests for multistage mechanisms (as the one modeled in this case study) is to proceed directly to each stage in turn, and continue the normal sequence from there.

In this example, I modify the initial model by introducing a fact to force the workflow of the process. For each action in the workflow, I have defined an Alloy fact that states which is the next action to be performed. As an example the following fact states that the action `ShowCatalog` must be followed by the action `ShowItem`:

```
fact {
all s: State, s': s.next{
ShowCatalog in s.action implies ShowItem in s'.action
  }
}
```

Another assumption that I make about the possible evolutions of the model is that each execution fragment must contain only one instance of every action. The Alloy fact that forces this behavior is:

```
fact {
  all s: State, s': s.next, x:Action{
    x in s.action implies x not in s'.^next.action
  }
}
```

In Section 4.2.2, the possible goal for the flaw that I am addressing in this section has not been presented. This is due to the fact that the goal that I am introducing here does not contain any information about any flaws but is only used in order to launch the model checking phase. Since the evolution of the model is forced by the facts introduced before, the following goal only checks if with a fixed numbers of sates, the action `ConfirmOrder` (as the last action of the workflow) can be "executed":

```
assert CheckWorkFlow {
  no s : State {
    ConfirmOrder in s.action  && s = last
  }
}
```

Attack trace(s)

The `check` statements used during the model checking phase and the resulting counterexamples are:

```
check CheckWorkFlow for 2 State
Trc0=[NoAction, ConfirmOrder]

check CheckWorkFlow for 3 State
Trc1=[NoAction, EnterDelivery, ConfirmOrder]

check CheckWorkFlow for 4 State
Trc2=[NoAction, EnterPayment, EnterDelivery, ConfirmOrder]

                          ...

check CheckWorkFlow for 8 State
Trc6=[NoAction, ShowItem, AddItem, ShowBasket, FinalizeOrder,
      EnterPayment, EnterDelivery, ConfirmOrder]

check CheckWorkFlow for 9 State
Trc7=[NoAction, ShowCatalog, ShowItem, AddItem, ShowBasket,
      FinalizeOrder, EnterPayment, EnterDelivery, ConfirmOrder]
```

As we can see, the traces show that the actions are selected with respect to the workflow with an increasing number of states.

Small conclusion

In this example, I show how a model can be used in order to derive attack traces where in the final state a security property is not invalidated. In this case the execution fragment itself can be seen as a possible attack trace since the workflow (that the developer wants a user to follow) of such traces is not followed. The main objection to this example is that the modeled web application is fictitious and thus a proper testing phase has not been performed. I believe that without an analysis of the results in testing real web applications with such approach, the results of the tests still require a validation by a security analyst.

5.5 Case Studies Conclusion

5.5.1 Models

In this section, I discuss the models of the case studies.

- WebGoat (I) - The general model of WebGoat discussed in Section 5.1.1 and depicted in Figure 5.1 has been used in order to test various lessons (for some of them a slightly modified version of the original model has been used). This model is well suited for our proof of concept since its behavior follows the one of a real web application where a user can login, manage a profile and see other users profile. The roles that are given to the users also help the testing of access control vulnerabilities.
- WebGoat (II) - One action models have been defined for different lessons of WebGoat. This choice reflects the structures of WebGoat (presenting an attack for each lesson even if it is composed only by one functionality). This type of models helps the proof of concept in showing the testing capabilities of the framework rather than its model checking phase.
- Gruyere - The Gruyere model (discussed in Section 5.2.1 and depicted in Figure 5.2) is composed by actions that permit users to add/delete messages (snippets), upload files, modify their profile, and see other user profiles. This variety of functionalities helps in (i) testing different vulnerabilities, and (ii) showing versatility of the framework in testing these vulnerabilities.
- DVWA - This model (discussed in Section 5.3.1 and depicted in Figure 5.3) follows the division into pages of the real web application (where every page contains a vulnerable entry point for a known vulnerability). From a model checker perspective the selection of the different actions during the model checking phase is quite simple. The variety of vulnerabilities makes it interesting for proving the testing capabilities of the framework.
- OnlineShop - This model (discussed in Section 5.4.1 and depicted in Figure 5.4) of a fictitious case study is used in order to derive counterexamples for logic flaws and thus proves that the model checking phase can be used also in this direction.

5.5.2 Types of Attacks

In this section, I summarize the results of the tests performed on the case studies dividing them by typology of attack. A tabular representation of the success of these attacks is reported in Table 5.5.

Access control flaws:

Table 5.5: Results of the tests performed in the case studies (Chapter 5). Legenda: CS – case studies; F – Framework, B – Burp Suite, Z – OWASP ZAP, and P – Paros; WG – WebGoat, GR – Gruyere, DV – DVWA, and OS – OnlineShop; ✓ – success of the tests, X – failure of the tests, ~ – ineffective test, and NA – the tests are inapplicable to the case study.

	CS	F	B	Z	P
Access Control Flaws					
Path Based Access Control Scheme	WG	✓	✓	✓	X
Presentational Layer Access Control	WG	✓	X	X	X
Data Layer Access Control	WG	✓	✓	~	X
AJAX Security					
DOM-Injection	WG	✓	X	X	X
Reflected XSS via AJAX	WG	X	✓	~	X
	GR	✓	X	X	X
Cross-Site Scripting					
Stored XSS	WG	✓	X	X	X
	GR	✓	~	X	X
	DV	✓	✓	X	✓
Targeted stored XSS	GR	✓	✓	X	X
Reflected XSS	WG	✓	✓	✓	✓
	WG	✓	✓	✓	X
	GR	✓	X	X	~
	DV	✓	✓	X	✓
File Upload XSS	GR	✓	NA	NA	NA
	DV	✓	NA	NA	NA
Injection Flaws					
Command-Injection (or Execution)	WG	X	X	X	X
	DV	✓	✓	X	X
SQL-Injection (string, numeric, . . .)	WG	✓	✓	X	X
	WG	✓	✓	X	X
	WG	✓	✓	X	X
	WG	✓	✓	X	X
	DV	✓	✓	X	✓
	DV	✓	✓	X	✓
Log Spoofing	WG	✓	✓	✓	✓
Insecure Configuration					
Forced Browsing	WG	✓	✓	~	X
	DV	✓	✓	X	X
Path traversal	GR	✓	X	~	~
Miscellaneous					
Password Brute Force	DV	✓	✓	~	NA
Logic Flaws					
Missing Checks	OS	✓	NA	NA	NA
Skip Stages	OS	✓	NA	NA	NA

- Path based access control scheme - The framework is able to access resources that are not referenced by a web application and thus automatize this type of attack.
- Presentational layer access control - The framework is able to test those functionalities that are accessible by users with high privileges (i.e., roles) using users with less privileges.
- Data layer access control - In this example, I introduced the possibility for the users of guessing data and thus augment the testing capabilities of the framework.

AJAX security:

- DOM-Injection - The framework is able to perform tests for blocked functionalities and try to bypass the restrictions coded in the DOM.
- Reflected XSS via AJAX - In one of the attacks, the implementation of the test execution engine has been proved to be flawed while it was able of discovering a vulnerability in another case.

Cross-Site Scripting (XSS):

- Reflected XSS - The framework has been able of finding the vulnerabilities.
- Stored XSS - The framework has been able to trigger XSS attacks (also on targeted data) in different locations from the ones used to deliver the payload.
- File upload XSS - The framework has been able of uploading a file that allows for the execution of an arbitrary script in the web application.

Injection flaws:

- Command-Injection (or execution) - In one of the attacks, the implementation of the test execution engine has been proved to be flawed while was able of discovering a vulnerability in another case.
- SQL-Injection (string, numeric, XPATH) - In the different instances of reflected SQL-Injection attacks the framework has been able of performing the tests with success.
- Log spoofing - The framework has been able of performing a log spoofing attack (combined with a XSS attack).

Insecure configuration:

- Forced browsing - The framework has been able of performing forced browsing attacks from given locations against the web applications under test.
- Path traversal - The framework has been able of performing path traversal attack.

Other vulnerabilities:

- Password brute force - The framework can be used in order to automatize brute forcing attacks.

Logic flaws:

- Missing checks - The framework has been used in order to test logic vulnerabilities where important checks are skipped by the web application.
- Skip stages - The framework has been able of deriving tests for multi-stage mechanisms.

5.5.3 Conclusions and Future Research Directions

As I have discussed in the previous sections, a variety of scenarios and attacks have been used during the initial proof of concept of the framework. In the following I discuss some conclusion for the different phases that compose the framework.

Modeling phase: As I have shown in the case studies, the behaviors that security analysts can model using the framework follow the ones of real web applications. The identification phase of the framework (i.e., identify which functionalities have to be modeled) has resulted to be simple and straightforward from the definition of the action. I believe that the examples of actions and their structure in the Alloy model (along with the definition of the transition system state) will help the security analysts in defining the actions for their analyzes. Moreover, the defined goals are general enough to be reusable in different models.

As future research directions I see the following:

- The definition of a larger database of actions (with the related parameters) will help the investigation on how the granularity of the actions (i.e., the different levels of abstraction that I can have in my models) changes the testing effectiveness of the framework.
- Since the set of atomic propositions has been proven to be satisfactory for the performed tests, an analysis on how to infer a model from a real web application could be of interest for future researches.

Concretization Methodology: The counterexamples derived from the model checking phase contain enough information to permit different concretization methodologies.

The methodology implemented in the preliminary version of the framework (i.e., counterexamples + hard coded configuration values and low-level definitions + instantiation level) has been able to deal with a variety of functionalities and types of attacks. The instantiation library can be extended with additional payloads without too much effort and thus is well suited to be customized by security analysts. Once a security analyst has developed suitable sets of payloads for the various attacks, the framework helps in her analysis by testing automatically the payloads for every possible entry point found for a given attack.

An investigation of the possible extensions in order to increase the testing coverage of the framework is also required. In this direction, an analysis on how to infer and gain data runtime from a web application should be made.

Test execution engine: The test execution engine has been able to perform a variety of tests for different vulnerabilities. The results of the test proved to be aligned with the results of the tests performed with the benchmark tools:

- The tests for well known vulnerabilities gave results comparable with the benchmark tools.
- In one case, the implementation of the test execution engine has proven to be flawed. The flaw was caused by a forced encoding on the data that the web application misinterpreted.
- In one instance, the security controls implemented on the server (not in the web application) blocked the attack.
- For two XSS attacks via file uploads and one instance of stored XSS, the test execution engine has proven to be efficient and successful in testing this type of attacks.

I believe that the small problems with the text execution engine can be solved with the extension of the framework with the VERA tool. As stated in Section 4.5.3, some adjustments are required on both the low-level attacker models and the python engine in order to permit the correct functioning of the VERA tool. Since the test execution engine suffers from some flaws (it has been used only as a proof of concept), a thorough analysis of the needed adjustments is required. During this analysis there is the possibility of having to reengineer the code of the test execution engine. On the other hand, with this extension the framework can be used by real testing group and helps analysis of web applications. One of the main advantages that the framework can bring in a testing environment is its usability as a valid alternative to other security tools (even though a certain experience is required).

Chapter 6

Related work

I have already mentioned a number of works in the areas of model-based and penetration testing, and I now discuss the differences between some approaches and the proposed framework.

6.1 Model-checking Driven Security Testing of Web-based Applications

One of the main problems of using model checking tools is to effectively test the implementation of the software be that a protocol, a program or, as in this thesis, a web application.

Considering protocols, an import line of work is presented in [4, 5]. While [4] focuses on how model checking could be applied to the security testing of web-based applications (e.g. SAML-based Single Sign-On for Google Apps), [5] focuses more on how to automatically concretize the counterexample produced by the model checker.

Models: The model used in [4, 5] describes the HTTP requests and responses that a client and a server exchange during the execution of a security protocol. The formal specification of a security protocol is used in order to define the model. After the model checking phase if counterexamples are found the implementation of the protocol is analyzed during a testing phase. In [4] the protocols are specified using HLPSL (the specification language of the AVISPA Tool [8]) while in [5] they are specified using ASLan [7] (a specification language developed in the context of the AVANTSSAR Project [6]).

The main difference that the model of the framework has with this approach resides in the fact that the level of abstraction of the model is different. A similarity can be found in the fact that both approaches are not interested in the implementation details of a protocol/functionality until the testing phase.

Model checking phase: In [4, 5], test cases are generated by checking if a security property (e.g., confidentiality, authenticity, authorization) of the model could be violated by a Dolev-Yao attacker.

Regarding the models, the major differences with this approach reside in:

- The goals used in the framework's models refer to known vulnerabilities.
- The model checking phase does not use an attacker entity during the analysis of a model.
- Since my approach uses known vulnerabilities, the model checking phase of the framework is used to derive those execution fragments that refer to known vulnerabilities. This means that new vulnerabilities (i.e., that have not been already discovered and formalized) cannot be found with my approach. Approaches like the one presented in [4, 5] can be used in order to building the set of vulnerabilities that are related to certain functionalities and can be tested with the framework.

Concretization methodology: In the specification of security protocols, the behavior of the agents involved in the communication is represented with abstract messages. The operations to check incoming messages and to generate outgoing ones are thus implicit.

In ASLan, message checks are realized by pattern matching procedures on fields:

- Received message must match some expressions stored in the state of the agent.
- Outgoing messages are calculated without specifying which operations are performed to compute it.

In order to interact with a system under test, they manually define the pattern matching procedures for these messages using a pseudo-language composed of statements such as *if-then-else*, *foreach* and the like. The test execution engine uses the same pseudo-language in order to execute the defined procedures and is thus specific to model checker but protocol independent. This means it could be applied generally to the analysis of different security protocols.

The difference in the concretization methodology used in the framework relies on the fact that the counterexamples that have to be concretized are on a different level of abstraction. The counterexamples for security protocols derived with the methodology of [4, 5] are more congruent to HTTP messages than the functionalities that I model.

Concluding, the methodology proposed in [4, 5] is focused on binding the specification of security protocols to actual implementations; the results are particularly promising but not directly comparable to mine, since my framework is at a different level of abstraction.

6.2 Mutation Testing

In [14, 15], the authors present an approach and a tool (SPaCiTE) for model-based security testing of web applications closely related to mutation testing.

Models: In [14, 15], the authors assume that security properties (e.g., confidentiality, authenticity, authorization) have to be provided with the model and that the model has to be secure with regard to these properties (i.e., if the model checker is run on this model no counterexample is found). The security analyst has to define the model by using abstract messages. These messages represent common actions a user of the web application can perform. The idea is that these abstract messages are sent to the server to tell it which action the client wants to perform. Models of web applications are defined using ASLan++ [7] (a language created for modeling security protocols). All users of the web application and the server are defined as entities. Since the model is created with a tool designed for security protocols it describes one possible interaction (the exchanged messages) between users and a web application.

Regarding the models, the major differences with this approach reside in:

- The model that I propose has not to be a secure model since my approach relies on the fact that counterexamples can be generated from a newly defined model.
- The goals used in the framework's models refer to known vulnerabilities while in this approach the goal is defined by the security analyst on the base of the content of the model.
- The conditions and operations that are used in order to define the functionalities of a web application describe both the client-side and the server-side of the application where the messages exchanged are not important.
- In my approach, defining the possible functionalities that can be accessed by a user while interacting with a web application do not require a user to follow a predefined sequence of actions (we give to the security

expert the freedom in deciding where the actions have to be concatenated).

Model checking phase: The generation the attack traces is made via mutation operators that are used in order to automatically introduce vulnerabilities in the model, i.e., invalidate the security properties. Mutants operators presuppose that standardized ASLan++ facts (i.e., facts like *check permission* or *has role*) have been used in order to define the model. Once a mutation operator has been applied to the model, the model checker verifies if the mutated model is still secure for the original goal. In this approach an implicit relation between the goals and the mutation operators is introduced: If the mutant operator is not in the scope of the goal (i.e., they use different facts) then the model results secure even if a vulnerability has been introduced.

Regarding the model checking phase, some differences reside in

- The goals defined for the framework are defined as representations of known vulnerabilities and are used in order to derive those execution fragments that permit to test known vulnerabilities. If a goal fails to find a counterexample then it will fail until new functionalities or data are introduced in the model.
- The goals defined for the framework can be used in different models (i.e., the goals are not model dependent).
- My approach does not mutate the model in the model-checking phase.

Concretization methodology: The authors propose a concretization methodology divided in two phases. In the first phase the WAAL language is used to map the exchanged abstract messages between agents into actions that a user performs in a web browser, i.e.,

- browser interface actions – which are similar to API methods from Selenium, and
- verification actions – which are used to verify whether an observed response matches with an expected one.

In the second phase the WAAL sequence of actions is mapped to executable source code (this mapping is done once and for all).

The difference in the concretization methodology used in the framework relies on the fact that the abstraction gap between the model and the implementation of web application is filled directly by using the HTTP requests that the functionalities use.

My framework provides to the security analyst the means to create a model of web application with actions that are used in all of the testing phases, whereas the approach in [14, 15] needs more expertise in its creation, and does not provide a standard methodology for doing it. On the other hand, the approach of [14, 15] is more mature than mine.

6.3 Formal Foundation of Web Security

[3] proposes a methodology for the analysis of several sample web mechanisms and applications, which relies on the Alloy analyzer but, again, resides on a different level of abstraction than my framework since they model network infrastructures rather than web applications.

Models: The modeling is devised to give the security analyst the ability of modeling network infrastructures and messages exchanged by entities connected to the network. The models describe what could occur if a user navigates the web and visits sites in the ways that the web is designed to be used. The model is composed by three parts:

- Web concepts that model the server and the browser.
- Threat models that define a web attacker and a network attacker.
- Security goal that define a security invariant and the session integrity.

The main difference between this approach and the framework is that the latter focuses on the interaction between the functionalities that compose a single web application, and how data types are handled.

Model checking phase: In [3], the authors identify two main security goals for web applications security:

- Security invariant - The web contains a large number of existing web applications that make assumptions about web security. They formalize these goal as a set of invariants expected to remain true during the model checking phase.
- Session integrity - When a server takes action based on receiving an HTTP request, the server often wishes to ensure that the request was generated by a trusted principal and not an attacker. They formalize this goal by recording the "cause" of each HTTP request and checking whether the attacker is in this casual chain.

Concretization methodology: In [3], the concretization methodology is not defined. A manual reproduction of the attacks found in the model checking phase is thus required in order to execute the test cases.

Chapter 7

Conclusion

7.1 Summary

In this thesis I have proposed a formal and flexible model-based security testing framework that supports a security analyst in carrying out security testing of web applications. During the analysis for the definition of my framework the principal goal has been to create a methodology that permits to reuse the expertise of the penetration testers in a model-based environment. A particular attention has been put in the simplification of the usage of model-based testing for those analysts who are not accustomed to such techniques.

The dissertation started with the definition of a suitable methodology for modeling web applications for security testing. I have introduced the formal definitions about the information that are modeled:

- The knowledge that users can acquire during the interaction with a web application, and the data that the users handle.
- The behavior that a web application can manifest through the interaction with a user.
- Those information that can be used in order to perform security testing.

These information have been used in order to define a state of the transition system that has been used during the analysis. I have completed the modeling approach by defining the actions (i.e., abstract representations of parts of the web application providing some particular functionalities) that permit the evolution of the transition system.

For concreteness the transition system has been instantiated as an Alloy model. I have defined the security goals (that the models have to satisfy) as flaws that a security analyst can tests and gave examples for a variety of vulnerabilities.

In order to fill the abstraction gap between the attack traces and the implementation of the web applications I have introduced a suitable concretization methodologies, and presented a preliminary version of the implementation of the framework.

As an initial proof of concept, I have shown the application of the framework to the four case studies WebGoat, DVWA, Gruyere and OnlineShop, and used three security tools as benchmarks for the various tests.

As illustrated by the case studies, the use of actions has a positive impact on all the phases of the framework that I propose, resulting, I believe, in a quite simple and flexible methodology for testing the security of web applications.

One of the strengths of my approach is the reusability of actions and of the sets U_{Knows}, WA_{Event} and $SEC_{Assertion}$ that compose the set of atomic proposition (AP, Definition 10). The expertise required in order to populate the InstLib is automatically "reused". The reusability of actions has also an implicit impact on scalability: the security analyst can identify existing parameters for the web application from a set of known actions (with associated parameters) or she can define new ones and then improve the set of actions.

If this strength may sound trivial for the more skilled penetration tester, I believe that the adoption of my framework by a testing group can result in a non-indifferent improvement for the testing capability of the whole group. New attack techniques and payloads can be inserted into the framework without changing (or compromising) the testing methodology.

Regarding the security goals, I am aware that a basic knowledge of logic is required in order to write them, but their reusability compensates the efforts put in their definition. As stated before, the security goals are a high-level representation of vulnerabilities, thus, a security analyst can use the same goal in models of different web applications without any problems.

To summarize, I believe that modeling web applications using actions, rather than using messages representing the underlying protocol, has a lot of potential but further work is, of course, still needed.

7.2 Future Work

Various future research directions are possible in all the phases of the framework. In the following I give an overview of the possible future work.

Modeling phase: Regarding the modeling phase the main task is to create a larger database of actions with the related parameters. Actions can be defined at some level of abstraction, thus multiple actions at different levels of abstraction can be added to the database for the same functionality (i.e., for a functionality one has to instantiate actions at different levels of abstraction).

With the definition of such database it is possible to investigate how the granularity of the actions changes the testing effectiveness of the framework while dealing with models with different levels of abstraction. This task will also give useful inputs on the possible extension of the remaining phases of the framework.

As an additional direction of research, the analysis on how to infer a model from a real web application could be of interest. Since the set of atomic propositions has been proven to be satisfactory for the performed tests, the idea is to search for the relations between the real web application's source code and the atomic propositions that have been defined. Even if interesting as a research direction, dealing with the source code of web applications is not a simple task. The variety of technologies used and the complexity of the code require a thorough investigation of the possible scenarios where the definition of these relations is possible.

Concretization Methodology: This phase of the framework is probably the one that requires more efforts in the future.

In this thesis, various attacks and flaws have been presented and tested. One possible direction is to investigate what extensions are required in order to increase the testing coverage of the framework, i.e., extend the possible attacks that the framework can test.

Another possible direction is to infer and gain data runtime from the web application under test. Implementing this feature in the framework has various benefits:

- The security analyst can interact with the framework only in those cases where the tests requires data that cannot be extracted from the web application.

- The framework can achieve a higher simplicity of use and at the same time lower the amount of work that is needed for a security analyst to test a web application with the framework.

Test execution engine: Since the test execution engine suffers from some flaws (it has been used only as a proof of concept), a thorough analysis of the needed adjustments is required:

- The test execution engine has to be able to use the different encodings that web applications use (e.g., HTML entity encoding, URL encoding or Unicode encoding), and leverage the possible outcomes that derive from the use of such encodings. The introduction of this feature requires an investigation on how to relate the different encodings with the functionalities that the test execution engine can request.

- For the browsing and check phases of the framework the python engine is implemented as a recursive function. In the check phase, the cookie management can be problematic when the test execution engine is dealing with multiple users. In order to solve this problem (i) an analysis of complex attack traces is required in order to derive the possible scenarios that can be obtained, and (ii) it possible that it will be necessary to reengineer the code of the test execution engine (i.e., the code of the functions that implement the browsing and check phases of the framework has to be rewritten).

Another promising direction of work is to introduce the VERA tool as test execution engine. Also in this case some adjustments are required:

- Since the low-level attacker models are defined to be used only by the VERA tool, using them within the framework requires one to introduce some ad-hoc functions in order to permit the correct execution of both the framework python engine and low-level attacker model itself.

- Also the python engine has to be modified in order to permit the correct functioning of the VERA tool.

Even if the first task could appear trivial, the introduction of the functions that manage the correct exchange between the execution of the python engine and the VERA tool is subordinated to the resolution of the flaws that the python engine has. Thus the introduction of the VERA tool as test execution engine has to wait until further analysis will be done.

Bibliography

[1] Alloy: A language & tool for relational models. `alloy.mit.edu/alloy/`, 2015.

[2] Acunetix. Acunetix web application security. `http://www.acunetix.com/vulnerability-scanner/`.

[3] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 290–304, Washington, DC, USA, 2010. IEEE Computer Society.

[4] Alessandro Armando, Roberto Carbone, Luca Compagna, Keqin Li, and Giancarlo Pellegrino. Model-checking driven security testing of web-based applications. In *ICSTW '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 361–370, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Alessandro Armando, Giancarlo Pellegrino, Roberto Carbone, Alessio Merlo, and Davide Balzarotti. From model-checking to automated testing of security protocols: bridging the gap. In *TAP'12: Proceedings of the 6th international conference on Tests and Proofs*, pages 3–18, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures. `http://www.avantssar.eu/`.

[7] AVANTSSAR. Deliverable 2.3 (update): ASLan++ specification and tutorial, 2011. Available at `http://www.avantssar.eu`.

[8] Automated Validation of Internet Security Protocols and Applications. `www.avispa-project.org`.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[10] Beef: The Browser Exploitation Framework Project. `http://beefproject.com/`.

[11] Philippe Biondi. Scapy. `http://www.secdev.org/projects/scapy/`.

[12] 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.

[13] Mugdha Bendre Bruce Leban and Parisa Tabriz. Gruyere: Web Application Exploits and Defenses. `https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`, 2015.

[14] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Semi-automatic security testing of web applications from a secure model. In *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012*, pages 253–262, 2012.

[15] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. SPaCiTE - web application testing engine. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, April 17-21, 2012*, pages 858–859, 2012.

[16] Steven R. Lavenhar Christoph Michael and Howard F. Lipson. Source Code Analysis Tools - Overview. `https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/263-BSI.html`, 2009.

[17] Chinotec Technologies Company. Paros - web application security assessment. `http://www.parosproxy.org/`.

[18] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: A systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, New York, NY, USA, 2007. ACM.

[19] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–207, 1983.

[20] Rik Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65 – 99, 2009.

[21] Ettercap. `http://ettercap.sourceforge.net/`.

[22] Firesheep. `http://codebutler.github.com/firesheep/`.

[23] HP Fortify. HP WebInspect. `https://www.fortify.com/products/web_inspect.html`.

[24] Grendel-Scan. `http://grendel-scan.com/`.

[25] Cenzic Hailstorm Professional. `http://www.cenzic.com/products/cenzic-hailstormPro/`.

[26] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, August 1976.

[27] IBM. Rational AppScan. `http://www-01.ibm.com/software/awdtools/appscan/`.

[28] Immunity Inc. Immunity CANVAS. `http://www.immunitysec.com/products-canvas.shtml`.

[29] Girish Janardhanudu and Ken van Wyk. White box testing. `https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box/259-BSI.html`, 2009.

[30] LAPSE: The Security Scanner for Java EE Applications. `https://www.owasp.org/index.php/OWASP_LAPSE_Project`.

[31] PortSwigger Ltd. Burp suite. `http://portswigger.net/burp/`.

[32] SensePost Pty Ltd. Wikto. `http://www.sensepost.com/labs/tools/pentest/wikto`.

[33] Gordon Lyon. Nmap security scanner. `http://www.nmap.org/`, 2011.

[34] Maltego. `http://www.paterva.com/web5/`.

[35] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[36] N-Stalker. N-Stalker Web Application Security Scanner. `http://www.nstalker.com/products/editions/`.

[37] Tenable network security. Tenalbe Nessus. `http://www.nessus.org/products/nessus`.

[38] OWASP. Open Web Application Security Project. https://www.owasp.org.

[39] OWASP. WebScarab. https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.

[40] OWASP. Zed Attack Proxy Project (ZAP). https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

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

[42] OWASP. OWASP Testing Guide. https://www.owasp.org/index.php/OWASP_Testing_Project, September 2014.

[43] OWASP. Broken access control. https://www.owasp.org/index.php/Broken_Access_Control, 2015.

[44] OWASP. Brute force attack. https://www.owasp.org/index.php/Brute_force_attack, 2015.

[45] OWASP. Command-Injection. https://www.owasp.org/index.php/Command_Injection, 2015.

[46] OWASP. Cross-site Scripting. https://www.owasp.org/index.php/XSS, 2015.

[47] OWASP. Forced browsing. https://www.owasp.org/index.php/Forced_browsing, 2015.

[48] OWASP. Injection Flaws. https://www.owasp.org/index.php/Injection_Flaws, 2015.

[49] OWASP. Path manipulation. https://www.owasp.org/index.php/Path_Manipulation, 2015.

[50] OWASP. Path traversal. https://www.owasp.org/index.php/Path_Traversal, 2015.

[51] OWASP. SQL-Injection. www.owasp.org/index.php/SQL_Injection, 2015.

[52] OWASP. Testing for AJAX Vulnerabilities. https://www.owasp.org/index.php/Testing_for_AJAX_Vulnerabilities_(OWASP-AJ-001), 2015.

[53] QualysGuard IT Security. http://www.qualys.com/products/qg_suite/.

[54] Rapid7. Metasploit Framework. http://www.metasploit.com/.

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

[56] SPaCIoS: Secure Provision and Consumption in the Internet of Services. www.spacios.eu, 2015.

[57] SPaCIoS: Definition of Attacker Behavior Models (Deliverable 2.4.1), 2012.

[58] SPaCIoS: Methodology and technology for vulnerability-driven security testing (Deliverable 3.3), 2013.

[59] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition.* John Wiley & Sons, Inc., New York, NY, USA, 2011.

[60] Chris Sullo and David Lodge. Nikto. http://www.cirt.net/nikto2.

[61] Nicolas Surribas. Wapiti. http://wapiti.sourceforge.net/, 2006.

[62] DVWA team. Damn Vulnerable Web App (DVWA). http://www.dvwa.co.uk, 2015.

[63] Core Security Technologies. Core Impact. http://www.coresecurity.com/content/core-impact-overview.

[64] Jennifer Tenzer and Perdita Stevens. On modelling recursive calls and callbacks with two variants of unified modelling language state diagrams. *Formal Asp. Comput.*, 18(4):397–420, 2006.

[65] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, August 2012.

[66] Kenneth R. van Wyk. Penetration Testing Tools. https://buildsecurityin.us-cert.gov/bsi/articles/tools/penetration/657-BSI.html, 2007.

[67] Wireshark. http://www.wireshark.org/.