Abstract
This deliverable contains descriptions of all selected application scenarios, including both industrial ones and public ones, describing the corresponding security goals, vulnerabilities and attacks. We provide formalizations for three application scenarios, along with a brief description of the prototype implementation. We discuss evaluation criteria for the SPaCioS methodologies and technologies, and report preliminary test and assessment results.

Deliverable details
Deliverable version: v1.0
Date of delivery: 03.10.2011
Editors: all
Classification: public
Due on: 30.09.2011
Total pages: 126

Project details
Start date: October 01, 2010
Duration: 36 months
Project Coordinator: Luca Viganò
Partners: UNIVR, ETH Zurich, KIT, INP, UNIGE, SAP, Siemens
Contents

1 Introduction 7

2 WebGoat 10
  2.1 Application Scenario Description ...................... 10
     2.1.1 Overview ...................................... 10
     2.1.2 Implementation Technologies .................... 10
     2.1.3 Architecture Description ...................... 11
  2.2 Security Goals .................................... 12
  2.3 Potential Vulnerabilities ............................ 14
  2.4 Potential Attacks .................................. 17
     2.4.1 Bypassing authentication systems ............... 17
     2.4.2 Bypassing authorization mechanisms ............ 17
     2.4.3 Violating availability goals ................... 18
     2.4.4 Violating integrity goals ...................... 18
  2.5 Formalization ..................................... 19

3 SAML 2.0 Web Single Sign-On 23
  3.1 Application Scenario Description .................... 23
  3.2 Security Goals .................................... 28
  3.3 Potential Vulnerabilities ............................ 28
  3.4 Potential Attacks .................................. 29
     3.4.1 Violating authentication goal ................... 29
     3.4.2 Bypassing authorization ....................... 30
     3.4.3 Violating integrity ............................ 30
  3.5 Formalization ..................................... 30

4 OpenID 36
  4.1 Application Scenario Description .................... 36
     4.1.1 Authentication Protocol ....................... 36
     4.1.2 Association Session Protocol ................. 36
     4.1.3 Signature Verification Protocol ............... 37
     4.1.4 OpenID Authentication Protocol with Session Association ................. 37
  4.2 Security Goals .................................... 38
  4.3 Potential Vulnerabilities ............................ 39
  4.4 Potential Attacks .................................. 41
5 OAuth 2.0 44
  5.1 Application Scenario Description 44
     5.1.1 Overview 44
     5.1.2 OAuth2 flows 45
     5.1.3 Protocol assumptions 48
  5.2 Security Goals 49
  5.3 Potential Vulnerabilities 50
  5.4 Potential Attacks 50
     5.4.1 Violating authentication 50
     5.4.2 Bypassing authorization 51

6 Pervasive Retail 52
  6.1 Application Scenario Description 52
     6.1.1 Overview 52
     6.1.2 Scenes Description 52
     6.1.3 Architecture Description 55
  6.2 Security Goals 58
  6.3 Potential Vulnerabilities 59
  6.4 Potential Attacks 60

7 Infobase Document Repository 62
  7.1 Application Scenario Description 62
     7.1.1 Overview 62
     7.1.2 Scenes Description 64
     7.1.3 Implementation Technologies 67
     7.1.4 Architecture Description 70
  7.2 Security Goals 71
  7.3 Potential Vulnerabilities 75
  7.4 Potential Attacks 76
  7.5 Formalization 77

8 eHealth 85
  8.1 Application Scenario Description 85
     8.1.1 Overview 85
     8.1.2 Scenes Description 87
     8.1.3 Architecture Description 90
  8.2 Security Goals 91
  8.3 Potential Vulnerabilities 94
  8.4 Potential Attacks 95
9 Evaluation Criteria for SPaClIoS Tools
9.1 Terminology .............................................. 97
9.2 Absolute metrics ......................................... 98
9.3 Comparative criteria ..................................... 100

10 Preliminary Assessment ................................. 102
10.1 Assessment of mutation-based testing within WebGoat . . . . 102
   10.1.1 Security Mutant .................................. 102
   10.1.2 Instantiation of the Abstract Attack Trace ........... 103
   10.1.3 Results and discussion ........................... 107
10.2 Assessment of test execution engine within SAML .......... 107
   10.2.1 Test results and conclusions ...................... 111

11 Conclusions .................................................. 112

A WebGoat WSDLScanning XML file .................. 113
B WebGoat ASLan++ code ................................ 115
C SAML ASLan++ code ..................................... 117
D Infobase ASLan++ code ................................ 120
List of Figures

1  Basic architecture of WebGoat ........................................ 11
2  Basic architecture of WebGoat Website Layout ...................... 13
3  WebGoat RBAC lesson organization chart ............................. 19
4  SAML SSO profile SP-initiated with front channels ............... 24
5  SAML SSO profile IdP-initiated with front channel ............... 25
6  SAML SSO profile SP-initiated with back channel ................. 26
7  SAML SSO profile IdP-initiated with back channel ............... 27
8  OpenID authentication protocol with Diffie-Hellman session association .......................... 38
9  Wrong use of secure transport layer .................................. 42
10 Abstract flow of OAuth2 ................................................. 45
11 OAuth2 Authorization Code flow ...................................... 46
12 OAuth2 Implicit Flow .................................................... 48
13 Scene 1: Individual Offer .............................................. 53
14 Scene 2: Coupon .......................................................... 54
15 High Level System Architecture ...................................... 56
16 Detailed System Architecture ......................................... 57
17 Offer Request ............................................................. 58
18 Assets, actions and stakeholders of the IDR system ............... 65
19 Possible scenarios, threats and risks associated with the IDR system ......................................... 66
20 Infobase login page ....................................................... 67
21 Infobase welcome page .................................................. 68
22 Repositories visible to Peter Smith .................................... 68
23 ABC Inc. Repository ..................................................... 69
24 ABC Inc. Repository after uploading the "Contract" document ....... 69
25 Component structure of the system ................................... 71
26 Servlet, interceptor filter and actions .................................. 72
27 Sequence diagram represents the interaction between the different components of the IDR system ......................................... 73
28 Infobase MSC ............................................................... 84
29 Basic architecture of an eHealth Server mashup ...................... 91
30 Basic architecture of an eHealth Client mashup ...................... 92
31 MSC of the AATr returned by the model checker .................... 109
32 MSC of the Google-SAML IATr ....................................... 110
1 Introduction

Work Package (WP) 5 of the SPaCIoS project defines a number of representative industry-relevant application scenarios. From these application scenarios, we extract problem cases that will be used to assess the effectiveness and efficiency of the validation techniques and tools developed in WP 3 and WP 4. We will undertake prototype implementations of the problem cases, which will be subject to a formal analysis. The subsequent assessment phase will evaluate the developed technologies according to a predefined list of criteria described in Section 9.

As a general remark, for this deliverable and actually for the rest of the project, note that we will adopt the following terminology: we will use application scenario as a synonym for case study (we prefer to speak of application scenario as it highlights the applicative nature of the example). Each application scenario will consist (or can be divided) into one or more scenes, where each scene gives rise to one or more problem cases, which is a formal description of (part of) the scene in order to capture a security aspect such as a security goal or a potential vulnerability. For example, in Pervasive Retail, we consider two scenes corresponding to different uses of the same infrastructure: the individual offer scene and the coupon scene. The actors and their relations may be different for different scenes. In the individual offer scene, in order to capture the security goal that the offer should be a secret between the Pervasive Retail Platform and the consumer, we can produce a formal description as a problem case.

The outcome of this WP is a proof of concept before the results of the project are migrated to industrial practice as described in WP 6. In this sense, this WP bridges between the conceptual level and the application level, providing input to real-world challenges as well as feedback on the practical suitability of the approaches and tools developed in SPaCIoS.

WP 5 comprises of two sub work packages: WP 5.1 “Application Scenarios and Prototyping” and WP 5.2 “Validation and Assessment”. In this deliverable, we report on our effort in both of the two sub work packages during the first project year. The work we report on here includes the following:

Description of the application scenarios. We identify and describe seven application scenarios. Four of them are from public domain:

- WebGoat is an interactive learning web application from OWASP (the Open Web Application Security Project), whose main goal is to “create a de-facto interactive teaching environment for web ap-
plication security” [17]. The deliberately insecure WebGoat application is structured in lessons: each lesson contains vulnerabilities that can be exploited through specific attacks.

- **SAML 2.0 Web Single Sign-On** is the emerging standard enabling on-line business partners to authenticate their users once within a federated identity environment.

- **OpenID** is an open and user-centric Web browser based Single Sign-On protocol. It provides a way to authenticate a user asking her to prove that she controls an identifier.

- **OAuth 2.0** is an authorization protocol aimed at granting to service $A$ the access to a protected resource (e.g., hosted on a service $B$) on behalf of a user, without requesting the provisioning of the user’s credentials to the service $A$.

The other three are proposed by the industrial partners:

- **Pervasive Retail** contains a new on-demand marketing management platform to create interactivity between consumers, retailers, and product providers through mobile phones to provide product and offer information to consumers. At the same time, the platform influences consumers at the point of decision and increases consumer traffic to retailers and in-store sales.

- **Infobase Document Repository** implements a Document Management System that allows for secure management and sharing of documents or data files using web browsers.

- **eHealth** is based on mashup systems that on one hand create and use electronic health records (private patient information) and on the other hand aggregate other functionalities, like decision support for the practitioner, analysis of images, billing systems, etc.

For each application scenario, in addition to descriptions of the application scenario concepts, functionalities, and prototype implementations, we identify also security related aspects including security mechanisms and goals, vulnerabilities, and potential attacks. A wide variety of such security aspects are covered by these application scenarios. These security related aspects form the basis of Deliverable D2.1.1 [22], in which these information are analyzed and summarized.

We have collected the security goals, potential vulnerabilities, and potential attacks by reading and analyzing documents and discussing with
relevant stakeholders of the application scenarios. For the time being, we consider these security goals, vulnerabilities and attacks to be relevant to the technologies and tools that will be developed in SPaCIoS project. During the execution of the project, some of them may become out of scope of SPaCIoS due to various reasons. Note that since making the application scenarios secure is not the ultimate goal of SPaCIoS, the security goals, vulnerabilities and attacks collected in this deliverable may not cover all the security related aspects in the application scenarios.

**Evaluation criteria.** We define criteria to evaluate the technologies and tools developed — and to be developed — in SPaCIoS.

**Preliminary assessment.** We report on the preliminary testing activities that we have performed to assess the technologies and tools developed in SPaCIoS so far.

The rest of this deliverable is organized as follows. Section 2 to Section 8 describe the seven application scenarios. Section 9 defines the evaluation criteria. Section 10 reports the preliminary assessments. Section 11 concludes the deliverable.
2 WebGoat

2.1 Application Scenario Description

2.1.1 Overview

WebGoat is an interactive learning web application from OWASP, the Open Web Application Security Project; it is listed as an OWASP project and is currently hosted on Google Code\(^1\). The main goal of the WebGoat application is to “illustrate typical security flaws within web-applications”\(^2\). The deliberately insecure WebGoat application is structured in lessons: each lesson contains vulnerabilities that might be exploited through specific attacks in the area of access control flaws, AJAX security, authentication flaws, concurrency, cross site scripting, SQL injection, parameter tampering, or session management flaws. This does not mean, however, that other attacks would not work against the specific vulnerability (e.g., missing input validation/output encoding also could allow for SQL-Injection) but only one attack is addressed per lesson.

Each lesson thus usually implements simple workflows, e.g., register/delete users. Workflows between the lessons are not connected and therefore no complex workflow exists within WebGoat. Nevertheless, lessons could easily be extended (or additional lessons could be created) in case more complex workflows should be required e.g., to address accountability.

2.1.2 Implementation Technologies

The following features characterize WebGoat:

- uses the Apache Tomcat servlet as an HTTP web server environment for Java;
- is based on J2EE, making use of JSON and XML;
- allows SOAP requests to be used to access application functionality (see the file WSDLScanning.xml in Appendix A);
- uses InstantDB\(^3\), a Java database, in the background for SQL injection related lessons;
- uses the Element Construction Set from the Jakarta\(^4\) project;

\(^1\)http://code.google.com/p/webgoat/
\(^3\)http://instantdb.tripod.com/
\(^4\)http://jakarta.apache.org

FP7-ICT-2009-5, ICT-2009.1.4
Project No. 257876
• uses the Eclipse Web Tools Platform\(^5\).

### 2.1.3 Architecture Description

In this section we describe the overall architecture of WebGoat, its main components, and how WebGoat can be extended.

![Basic architecture of WebGoat](http://www.eclipse.org/webtools)

- WebGoat is a web-based testing platform for typical security assessments. It consists of a core application that can be extended by lessons. As described in Figure 1 [14, page 10], the overall architecture of WebGoat follows the model-view-controller approach. The controller is implemented as Java servlets whereas the view consists of JavaServer Pages. The model is realized by different ActionHandlers and a WebSession.

  WebGoat provides a large — and growing — number of lessons. In order to successfully solve the lessons, third party applications are needed as well. For instance, a standalone web proxy or a browser proxy plugin will be required that is able to process and forward HTTP and HTTPS traffic. Such a web proxy must therefore sit between the client application (browser) and the WebGoat server application. In addition, the

\(^5\)http://www.eclipse.org/webtools
web proxy should be able to display and modify all data in transit. If a model about a lesson is incomplete, Web application spiders can be used to gather information about the structure of the lesson.

- Adding a new lesson to the WebGoat core application is rather simple, since the application provides so called LessonAdapters so that new lessons can easily be integrated into WebGoat. The Java class of a new lesson inherits from the LessonAdapter and implements the necessary methods. The most important method is the “createContent” method that has to handle each request of the user and the next screen that has to appear.

- When a lesson is executed, the action handler executes the business logic of the lesson and loads data into the WebGoat WebSession object. Finally, it passes the control over to the view component. In order to track the status of the lesson, a lesson tracker objects is used and updated.

- The front-end of WebGoat shown in Figure 2 [14, page 11] consists of the view component, whose main elements are a Main Header, a Navigation Bar, a Lesson Header and a Lesson Content. When it comes to the output of a lesson only the lesson content is allowed to be changed. Furthermore the lesson content may only consist of a single page (and thus each lesson should be handled on a single page).

### 2.2 Security Goals

WebGoat lessons are grouped in categories. Each category addresses a specific kind of vulnerability or attack (access control flaws, AJAX security, authentication flaws, cross-site scripting, etc.). At the time of writing this deliverable, there were almost 60 lessons organized in 17 categories.

Even though WebGoat lessons are vulnerability-oriented and do not explicitly specify which kind of security property is violated, we list here security properties that can be tested in at least one of the WebGoat lessons.

**Authenticity** Two lessons in WebGoat address directly authentication problems due to the use of basic protocols: guessing a secret color (brute force attack), or sending credentials encrypted with invertible functions like base64. While finding those low-level attacks is a major issue, SPaCIoS aims at detecting complex attacks at the protocol level and generally assumes a system based on perfect encryption functions. Thus, finding those low-level errors is
out of SPaCIoS scope. Nevertheless, even though the other WebGoat lessons do not directly address authentication flaws, they do rely on a access control system to ensure other security goals. If one can bypass an authentication system, it may also violate another security property as confidentiality for example. Thus, it is also important to consider authenticity properties in ASLan++ models of WebGoat lessons. Authenticity goals are built-in primitives in ASLan++. 

**Authorization** Several lessons aim at bypassing a role-based authorization system to access to confidential data from an unauthorized user. For example, in the stored cross-site scripting (XSS) lesson, a user edits his profile to insert a script that will be executed by everyone viewing his profile. Thus, it is possible for this user to become one of the user that has access to his profile and then get access to other profiles he should not have access to. There is no built-in primitive in ASLan++ to express authorization goals but they can be modeled using predicates and Horn clauses (or using LTL formulae if the rights change over time). Note that expressing goals as LTL formulae is more difficult for a security analyst than using built-in primitives like for confidentiality and authenticity.
**Availability** A lesson deals with Denial-of-Service (DoS) attacks, which is the main vulnerability against availability security goal. In this lesson, the server should authorize at most two connections at the same time. However, due to a race condition flaw, it is possible to log in with more than two users in parallel. Using this flaw can lead to a DoS attack by overloading the server bandwidth, which is probably designed for at most two users according to the defined limitation. Defining availability goals at the ASLan++ model is out of SPaCIoS scope [22, §3.7].

**Confidentiality** In most of WebGoat lessons, the aim is to find a way to view confidential data. The ways to achieve this goal are extremely various: bypassing the authentication request, sniffing user credentials or stealing a user session to get access to his confidential data, injecting (SQL or XSS) code to lead either the server or a user to reveal confidential data. Defining secrecy goals is one of built-in primitives available in ASLan++.

**Integrity** Certain values must not be altered in an unintended/unauthorized way. For example, if a test is performed on a value x and if the test passes a certain action is performed. It should not be possible for an attacker to change the value of x between the test and the action execution. This notion of atomicity is crucial in Web application and its violation can lead, for example, to race condition flaws. Two lessons in WebGoat deal with race condition flaws that can be exploited to violate an integrity goal.

### 2.3 Potential Vulnerabilities

In this section we provide a list of different potential vulnerabilities that are addressed by WebGoat lessons. This list of vulnerabilities is limited to the ones that can be tested in at least a WebGoat lesson but a more complete list is available in the OWASP testing guide⁶.

**Authorization flaws** Access control aims at ensuring that confidential data are restricted to authorized users only. If the access control is enforced by presentation layer only (i.e., hiding buttons or links that are not authorized to the user), then it is easy for an intruder to directly access to the confidential data. An example in WebGoat of this vulnerability can be found in “WSDL scanning” lesson, where a function from the WSDL document is not visible inside the user Web interface but still executable. The “Bypass

Business Layer” and “Bypass Data Layer” lessons are also examples of access control enforced by presentation layer only, where it can easily be circumvented by a forced browsing attack (by modifying either the action or the employee_id field inside a POST request).

Path traversal is another attack that also exploits an authorization flaw (for example the “dot-dot-slash” vulnerability). The “Bypass a Path Based Access Control Scheme” lesson shows how a path traversal attack can view configuration files that are outside the user directory.

**Broken authentication** Authentication is usually the first part of a Web application seen by a user. She has to prove her identity (by providing credentials) in order to access to her confidential data. If those credentials are transported over a non secure channel they may be intercepted by a malicious user that can reuse them afterward. Other common flaws on an authentication system are the use of a weak password that can be guessed or offering a vulnerable remember password system like “What is your favorite color?”, where the number of possible answers is very limited (e.g., in “Forgot Password” lesson).

**Broken session management** HTTP is a state-less protocol. It is then necessary to rely on an application-specific session management system in order to connect the different HTTP messages with each other. A Web application may have flaws in its session management system such as: not checking if a cookie is valid, using a weak session identifiers, reusing an existing session identifier when authenticating a user, or not checking if the maximal number of sessions that can be opened is reached. The last vulnerability can lead to a DoS attack while the former ones can be exploited for bypassing the access control system or using the session of an authenticated user. “Hijack a Session”, “Spoof an Authentication Cookie” and “Session Fixation” lessons deal with session management flaws.

**Concurrency flaws** Usually web applications are designed to manage multiple connections at the same time. Thus several requests from many users can be addressed in parallel. However, race condition flaws can occur in such multi-threaded applications and then produce an unexpected behavior, which could be exploited by a malicious user to violate integrity properties but also other security goals like confidentiality. Two lessons in WebGoat deal with concurrency flaws, namely “Thread Safety Problems” and “Shopping Cart Concurrency Flaw”.

---

FP7-ICT-2009-5, ICT-2009.1.4
Project No. 257876

**SPaCiO**
Information leakage  The first step of a pen-tester is to gather information about the system under test. Thus, if some confidential data are left out inside HTML comments (e.g., user name/password similar to leave around a sticky note), or if error messages reveal sensitive information (e.g., disclosing the application content), then these vulnerabilities can be exploited by a malicious user to compromise the application security: privilege escalation, forced browsing, etc. The “Discover Clues in the HTML” lesson is one example of exploiting information leakage.

Improper error handling  Another crucial place where flaws can occur is at the error handling part of the system. For example, in the corresponding WebGoat lesson about improper error handling, a catch block (triggered when an exception due to a missing field in a form occurs) leads to a successful log in even if the given credentials are not valid.

Lack of input validation  An improper validation of received data can lead to various attacks like injection such as XSS, SQL, XML/JSON and XPATH injection, or path traversing attack, or also taking advantage of unused operations announced by a WSDL document. Several lessons in WebGoat exploit a lack-of-data-validation vulnerability to violate security properties like confidentiality or authenticity.

It is also important to validate (for example by signing) output data so an intruder that would intercept a message cannot modify it without being noticed by the client. Thus, an improper output validation can lead to an integrity violation. See for example the “DOM Injection” lesson in WebGoat, where a HTTP response is tampered to inject a JavaScript command that will be executed by the client’s browser.

Security misconfiguration  Most applications, frameworks and operating systems provide security protections. However, these systems usually need to be configured correctly in order to activate such protections. For example, the default account passwords must be changed or deactivated and more generally, everything unnecessary should be removed (e.g., ports, services, privileges). It is also important that every part of the system be up to date in order to fix known security flaws. Achieving this correct configuration is far from being obvious and requires a certain knowledge about each component of the system. Thus, there are several vulnerabilities that are due to security misconfiguration. For example, the WSDL scanning lesson in WebGoat shows a violation of the minimal exposure principle. In that lesson, more operations are present in the WSDL document than the operations used by
the Web service. Thus, a malicious user can exploit these extra operations to disclose assets that are not visible from a normal use of the Web service.

2.4 Potential Attacks

In this section we provide a list of different potential attacks that are present in WebGoat lessons.

2.4.1 Bypassing authentication systems

**Brute force attack** In a WebGoat lesson, an intruder has to guess a color to retrieve a user password. For this kind of limited space search, a brute force attack that consists in trying all different values is practical. As another example, Web applications usually save hash value of user passwords inside a database. If those values are revealed, it is possible to find out some user passwords thanks to rainbow tables, especially if no specific salt value is added by the application. Those attacks exploit low-level cryptographic function flaws and, thus, are outside the scope of SPaCIoS.

2.4.2 Bypassing authorization mechanisms

**Forced browsing** A normal user follows the links proposed by the pages of a web application but an intruder can forge any HTTP request and so does not need to follow the structure proposed by the Web application. By doing such “forced browsing” attacks, a malicious user can exploit vulnerable web applications that assumes a user will never try to access to hidden parts of the application or will always log in before accessing to a protected asset. For example, in the lesson about role-based access control, a user cannot delete a profile unless he is a privileged user (the delete link does not appear on the list of profiles). However, the system forgot to check if the requested action is allowed. Thus, an attacker can bypass the access control mechanism by directly requesting the deletion of a user profile. In terms of a control flow graph of a system, this attack behavior is equal to a “jump” from one state to another where there is no link between those two states.

**Injection** Injection such as SQL, WSDL and XPATH injection, is one of the most found attacks against Web applications. XSS are a family of (java)script injections that target the web application and the client browser, but there are other injections that target one of the application backends as for example a database with SQL injection. In case of HTTP splitting, HTTP injection is performed by being able to insert CR+LF sequence inside the
header of an HTML response. WebGoat lessons address all of these injection attacks.

**Stored cross-site scripting** An application is vulnerable to stored cross-site scripting (XSS) attacks if and only if: a field used for creating or updating a datum D is not correctly sanitized and this datum D is used later without correctly sanitize its display. If a data-flow model [10] for the application is available, detecting stored XSS can be achieved by testing if every def-use path associated to a datum D is XSS safe.

**Reflected cross-site scripting** In contrast to a stored XSS, a reflected XSS does not need to be stored on the vulnerable website. It is enough to send the script by email to the victim. If the victim is already connected to the vulnerable website when receiving the email, the attack will be triggered as soon as she opens the malicious script.

**DOM-based cross-site scripting** The vulnerable web application is involved in the loop of both stored and reflected XSS attacks. This is not anymore the case with DOM-based XSS. Here the attack targets the post-formatting done by the client’s browser. Thus, a DOM-based XSS concerns the client and his browser only. Every XSS mitigation installed on the web application is ineffective against that.

### 2.4.3 Violating availability goals

**Denial of service (DoS)** By using a race condition flaw in WebGoat, one can connect to an application with as many users as he wants even though the application should be limited to two users in parallel. Thus, it makes a DoS attack possible by overloading the server bandwidth.

### 2.4.4 Violating integrity goals

**Time and state attacks** Two lessons deal with race condition flaws where accessing the same resource at the same time may produce an unexpected behavior. A concurrency flaw may break a confidentiality goal by revealing a content intended for someone else but in general it violates an integrity goal. For example, the goal of a WebGoat lesson is to change the price value for a list of articles in a merchandise web application. The presence of this vulnerability in a Web application is due to the use of variables that are not thread safe. This kind of vulnerability is usually difficult to find out by manual penetration testing as it implies to simultaneously accessing the
same resource. However, model-checker are well suited to detect concurrency problems and thus SPaCIoS tool should be better in detecting concurrency flaws than classical penetration test tools.

WebGoat lessons cover most of the common vulnerabilities that can be found in a Web application. Usually, each lesson focuses on a specific attack that exploits a specific vulnerability. Thus, this application scenario is well suited to evaluate which vulnerabilities can be detected by the SPaCIoS tool. However, no model is provided for the lessons nor even informal specification. Thus, we have to deduce the specification by browsing the WebGoat lesson and come up with security goals and a safe model for each lesson.

2.5 Formalization

As an example we consider a WebGoat lesson about authentication flaws. This lesson consists of a Web application that deals with user profiles and is used by different groups of people (e.g. admins, employees, managers, HRs). After log in, a user can view different profiles where the set of viewable profiles may differ from user to user. The user gets an HTML page where the profiles are displayed in a list element and the different actions are represented as HTML buttons. Access control to profiles is realized using role-based concepts. Each user is assigned to a role and can view profiles according to his role (see Figure 3). At the implementation level, access to a profile is

![WebGoat RBAC lesson organization chart](image)

Figure 3: WebGoat RBAC lesson organization chart
granted by adding the profile to the HTML list so that it can be selected. Access is denied by removing the profile from the list so that it cannot be selected anymore. Finally the selected profile may be displayed by clicking on the `viewProfile` button. All this functionality is implemented at the client side.

```plaintext
entity Environment {
    symbols
    login(agent, symmetric_key) : message;
    viewProfileOf(agent) : message;
    nonpublic noninvertible password(agent, agent) : symmetric_key;
    % used by Server entity to store users' cookies
    nonpublic cookies(agent) : (agent * cookie) set;
    canView(agent, agent) : fact;
    % ...}
```

Listing 1: Symbolic function definitions in the Environment model

Listing 1 shows how the Web application actions like `login` or `viewProfileOf` are represented by symbolic functions in ASLan++. The meaning of `password` and `cookies` functions will be explain later. In addition, the definition of `canView` fact is used to express the access control mechanism as Horn clauses in ASLan++.

```plaintext
entity User(Actor, S : agent) {
    symbols
    A : agent;
    Profile : profile;
    Cookie : cookie;
    body { % of User
    % login
    Actor --> S : login(Actor, secret_pvd:(password(A, S )));
    S -->* Actor : secret_cookie: (?Cookie); % user gets his cookie
    % viewProfile
    if (Actor -->canView(?A)) {
        Actor -->* S : Cookie.viewProfileOf(A);
        S -->* Actor : A.?Profile;
    }
}
```

Listing 2: User agent model

To formally describe the previously introduced web application we specify two different agents that communicate with each other. The communication is modeled at the HTML level and represents the important steps a user has to perform to view a profile. The sequence of messages that are exchanged between the user and the server consists of four messages. In the first step the user sends his credentials to the server which is modeled by the abstract message `login` (see Line 18 of Listing 2). In this scenario the two agents communicate over a encrypted but non-authenticated channel, expressed by the channel sign `-->*`, or over a encrypted and authenticated channel, with
the *->* notation. As credential, he uses a password that is modeled as a nonpublic and noninvertible function (see lines 5 and 18), so the user password is a shared secret between the user and the server (this security goal is defined Line 28).

```
goals
  secret_pwd:( _) {U,S};
  secret_cookie:( _) {U, S}; % the set of agents who may know the cookie
```

Listing 3: Security goals that belong to a Session between a user U and a Web server S

If the server accepts the credential it sends back a secret cookie in the second step of the communication. Steps three and four are then about viewing the profile. The user can choose any profile he is allowed to view: he send the `viewProfileOf` message together with the previously received cookie. Finally the requested profile is sent back to the user.

```
entity Server(U, Actor : agent) {
  body (% of server
    while(true) {
      select { % the server acts upon receiving messages from a client
        % Login action
        on(?U ->* Actor: login(?U, secret_pwd:(password(?U, Actor))): {
          % Create a new cookie for this user
          secret_cookie:(Cookie) := fresh();
          % Store this relation
          cookies(Actor)->contains((?U, Cookie));
          % Send the cookie back to the user
          Actor *->* U: Cookie;
        }
        % ViewProfile action
        on(?U ->* Actor: ?Cookie.viewProfileOf(?A)
          & cookies(Actor)->contains((?U, ?Cookie))
          & ?U->canView(?A)
        ): { % Comment out the next line to exhibit an attack
          select { on(A->hasProfile(?Profile))(): } % get the profile of A
          Actor *->* U: A.Profile;
        }
      }
    }
  }
```

Listing 4: Server agent model

The model of the server agent (Listing 4) mainly consists of a list of valid messages. For the above described scenario two messages are important, the `login` and `viewProfileOf` message. If a user agent sends a login message, the server agent creates a new fresh cookie, binds it to the user agent and sends the cookie back to the user agent. If the server agent receives a `viewProfileOf` message from user agent U the server checks whether the cookie contained in the request is bound to the same user agent U (Line 45). In addition it is
checked if user agent $U$ is allowed to view the requested profile (Line 47). At the modeling level this is specified by fact statements. If access is granted the requested profile is sent back to user agent $U$.

56 goals
57 secret_profiles:
58  forall A P. [](iknows(P) & A->hasProfile(P) => i->canView(A));

Listing 5: Security goals of the Web application

Finally, security goals are described. Actually, we have seen intermediate security goals in Listing 3 that belong to a user session. Additional goals are defined in the Environment entity, where it is possible to specify goals according to the known environment (e.g. the users and their rights). In Listing 5, we define the following authorization goal: a malicious user cannot access to a profile he should not be allowed to view.

59 dishonest(curly);
60 iknows(inv(ak(curable)));
61 iknows(inv(ck(curable)));
62 iknows(inv(pk(curable)));
63 i->canVisit(curable);
64 iknows(password(curable, webServer));
65 iknows(curableProfile);

Listing 6: Curly is a compromised user

Note that in ASLan++, there is a default malicious agent, called $i$. However, in the current WebGoat lesson, the attacker should play the role of curly. Thus, curly is described as a compromised agent in our model so the intruder has access to her knowledge (in Listing 6, curly is declared as dishonest and all her secrets are transferred to $i$: her password, her secret keys, her secret profile and her rights to other profiles).

At the end, the model is consider as secure as no violation of a defined security goal is found by a model-checker. The full ASLan++ model of this WebGoat lesson is available in Appendix B. We also conducted preliminary assessment using this model and the results are presented in Section 10.1.
3 SAML 2.0 Web Single Sign-On

3.1 Application Scenario Description

Nowadays companies providing on-line user-centric services want to federate their business with new partners in order to offer an enriched set of services to their customers. Federation requires to share data and users.

The OASIS Security Assertion Markup Language 2.0 [7] Web browser Single Sign-On (SAML SSO, for short) is the emerging standard in this context enabling on-line business partners to authenticate their users once within a federated identity environment.

SAML SSO is a standardized, open, and interoperable solution. In this respect, it offers a significant number of configuration options allowing it to be applicable in a multitude of environments. This is the main reason of its adoption by SSO solution providers such as SAP. It is based on an XML format for encoding security assertions as well as a number of protocols and bindings that prescribe how assertions should be exchanged in a variety of applications and/or deployment scenarios. Three roles take part in the protocol: a client C, an identity provider IdP and a service provider SP. The objective of C, typically a web browser guided by a user, is to get access to a service or a resource provided by SP. IdP authenticates C and issues corresponding authentication assertions (a special type of assertions used to authenticate users). An authentication assertion contains, among others, a value called SessionIndex. This value is part of the global user session that is used in later stages to reverse the SSO process. The SSO protocol ends when SP consumes the assertions generated by IdP to grant or deny C access to the requested resource.

A SAML SSO profile offers two main usages depending on whether the web user requests a resource from an SP by contacting the SP directly (SP-initiated SSO), or by contacting the IdP that presents a set of SP resources that web users can consume (IdP-initiated SSO). Both SP-initiated and IdP-initiated SSO can be used in combination with the artifact resolution protocol (ARP) that provides a mechanism by which SAML messages can be transported in a SAML binding by reference instead of by value. In addition, SAML SSO offers many configuration options ranging from optional fields in messages, usage of SSL/TLS at transport layer, application of encryption and digital signature on specific sensitive message fragments, etc.

In the sequel of this section we detail both SP-initiated and IdP-initiated SAML SSO variants with and without artifact resolution. Use of artifact resolution is often referred to as back channels, while front channels indicates that artifact resolution is not used.
SAML SSO SP-initiated with front channel  Figure 4 shows the reference flow for the SAML SSO SP-initiated variant with front channels (i.e., without artifact resolution). In step S1, C asks SP to provide the resource located at the address URI. SP then initiates the Authentication Protocol by sending C a redirect response (e.g., HTTP 302 Response message) directed to IdP containing an authentication request of the form \( \text{AuthnReq}(\text{SP}, \text{IdP}, \text{ID}) \) where ID is a string uniquely identifying the request. IdP then challenges C to provide valid credentials and if the authentication succeeds IdP builds an authentication assertion \( \text{AA} = \text{AuthnAssert}(\text{SP}, \text{IdP}, \text{C}, \text{ID}) \) and places it into a response message \( \text{Resp} = \text{AuthnResp}(\text{ID}, \text{SP}, \text{IdP}, \{\text{AA}\}_{\text{K}^{-1}_{\text{IdP}}}) \), where \( \{\text{AA}\}_{\text{K}^{-1}_{\text{IdP}}} \) is the assertion digitally signed with \( \text{K}^{-1}_{\text{IdP}} \), the private key of IdP. SAML does not prescribe how the IdP authenticates C. This is thus abstracted away from our formalization and analysis where we simply assume that a successful user authentication takes place.

IdP then places \( \text{Resp} \) into an HTML form and sends it back to C (SAML POST Binding). The response is forwarded by using a client-side script that triggers the POST submission to SP. This completes the message exchange.
and SP can deliver the requested resource to C.

**SAML SSO IdP-initiated with front channels**  The message flow is shown in Figure 5. In this variant, IdP presents to clients resource links to services provided by SPs belonging to its federated environment. Instead of visiting SP directly, C asks IdP to access SP’s resources (step S1). Once C authenticates with IdP, IdP initiates the SAML Authentication Protocol by building an authentication assertion as already seen in the SP-initiated variant.

**Back channels (Artifact Resolution)**  While in Figure 4 and Figure 5, SAML messages are exchanged by value through C acting as intermediary, SAML also defines another method for exchanging assertions, not as a value but as a reference via usage of *Artifacts*. The idea is very simple: authentication requests and/or responses are internally stored by the issuer and a reference to this (aka the Artifact) is sent. The consumer of this artifact will then have to resolve it by directly contacting the artifact issuer without
Figure 6: SAML SSO profile SP-initiated with back channel

intermediary using back channels.

Figure 6 shows an SP-initiated SSO with back channels used on the SAML response message. The flow is like in Figure 4 until IdP receives the authentication request (step A2). IdP then issues the authentication assertion AA and stores internally a message AuthnResp(ID, SP, IdP, {AA}K_{IdP}^{-1}) (henceforth denoted Resp). Then it creates an Artifact = (IdP, REF), where REF is the internal pointer for Resp. The Artifact is sent to C, asking to forward it to SP. Once SP is in possession of Artifact (step A4), it issues an ArtResolve(IDa, SP, Artifact) where IDa is a string uniquely identifying the resolve request (step A5). The resolve request is sent over back channels to IdP. IdP then fetches Resp, encapsulates it in a ArtResponse(IDa, Resp) and sends it back to SP. At the end of the run, SP serves the resource to C.

Similarly, when enabling back channels in the IdP-initiated profile, SP receives an artifact Artifact from IdP in step A3 of Figure 7. Then IdP will resolve it transmitting the authentication response to SP.
Figure 7: SAML SSO profile IdP-initiated with back channel

Secure transport layer  The SAML 2.0 specifications repeatedly state the following assumptions of the transport protocols used to carry the protocol messages:

- Communication between C and SP can be carried over a unilateral SSL/TLS channel, established through the exchange of a valid certificate (from SP to C).

- Communication between C and IdP is carried over a unilateral SSL/TLS channel that becomes bilateral once C authenticates itself on IdP. This is established through the exchange of a valid certificate (from IdP to C) and of valid credentials (from C to IdP).

SAML SSO is an highly configurable protocol with several implementations available (in terms of APIs and frameworks) and we would not like to bind our analysis to specific systems. However, we targeted real systems as shown in Section 10.2 below.
3.2 Security Goals

Authentication Authentication involves C and SP entities. In general, we define that X authenticates Y on Z iff at the end of the protocol run X believes it has been talking with Y and they agree on the value of Z. In this specific case, we define a mutual authentication security goal:

- Client authentication: SP authenticates C on URI.
- Service provider authentication: C authenticates SP on Resource.

We would like to check whether SAML SSO ensures that if an user C accesses to a SP’s resource within a user session represented by a token sid (e.g. an HTTP cookie), then an IdP issued an assertion for C to SP for the same sid and sid is still valid.

Confidentiality We distinguish between two kinds of confidentiality goals:

- Confidentiality of resource: The resource provided by SP to C at the end of the protocol run (Resource) must be kept secret between the two entities; the address of the resource (URI) must be kept secret among C, SP, and IdP.

- Confidentiality of Authentication/Authorization data, namely authentication requests and assertions.

Integrity It must be granted both on the resource provided by SP and on assertion request/response. Regarding the first case, when C asks for a resource, it should not be possible to change the obtained resource without C being able to notice this change. Similarly, assertions must be checked against modifications that can occur during the exchange of requests/responses between the involved entities.

3.3 Potential Vulnerabilities

Potential vulnerabilities in SAML implementations can be restricted to:

Input validation vulnerabilities The validation of input is a mandatory requirement in SAML SSO specification. Actual systems that implement the SAML-SSO protocol and do not provide proper input sanitization can be weak to Cross-Site Scripting (XSS) and Cross-Site Request Forgery (XSRF) attacks.
Path vulnerability  In SSO protocol, the client requires the access to a resource located at a URL on a SP. Depending on the format of the URL, some information can be obtained by analyzing the URL string, thus revealing, for instance, the organization of part of the file system on the SP. Consequently, attacks based on path traversal can be tried out on the SP. The implementation of actual systems should require the use of URL that do not provide any information on the file system of the SP.

Insecure Transport vulnerability  An actual system where assertions are exchanged on an insecure channel can suffer from session management vulnerability where an attacker can steal the assertion and then impersonate the client, accessing the reserved resource.

Protocol Errors vulnerability  The possibility to accept a self-reported DNS name in an actual system can allow the exploitation of the SAML SSO Authentication Flaw (see [2]) where the client can be redirected to a malicious SP instead of the correct SP, through a wrong DNS resolution.

3.4 Potential Attacks

Attacks on the actual system which can violate the security goals by exploiting the previous vulnerabilities.

3.4.1 Violating authentication goal

Reflected cross-site scripting  Since C can firstly interact with a malicious SP, the URL originally requested by C can be tampered with a maliciously-crafted URL or injected with code that can be executed on the honest SP.

Cross-site request forgery  An actual implementation can suffer from an XSRF vulnerability, whether the resource requested by C from SP is not a physical one but rather an URL-encoded command, such as a request for the change of some settings or user’s preferences, or a command for deleting a resource, commitment of an action and so on. The malicious SP can force the execution of a command on the legitimate SP as it would be requested by C. The output of this action is provided to C. However, if the execution of the command does not provide any output to C, then the exploitation of the XSRF by the malicious SP can be fully unnoticed.

This attack can be realized by exploiting input validation, protocol error and reflected XSS vulnerabilities.
3.4.2 Bypassing authorization

**Broken authorization** The SP can retrieve information by analyzing the URL of the resource requested by C. From the analysis of the path, the malicious SP can attempt to violate access control by attempting a Path Traversal attack for accessing the file system of the honest SP. This point is specifically related with the web application, rather than SAML SSO protocol.

This attack can be realized by exploiting the *path traversal* and *insecure transport* vulnerabilities.

3.4.3 Violating integrity

**Delivery of unrequested resource** An SP could mislead the client to receive a resource different from the one originally requested. Moreover, the same SP can redirect the client to a legitimate SP before the beginning of the SAML SSO protocol run. In both cases, the client is provided with an unrequested resource. Some browser-side plug-ins can mitigate this problem by restricting HTTP redirections.

This attack can be realized by exploiting the *reflected XSS* and *insecure transport* vulnerabilities.

3.5 Formalization

In this subsection we present a formal model of SAML SSO SP-initiated with front channels for what concerns the client authentication security goal (see Section 3.2). Other SAML SSO variants and security goals will be considered in the forthcoming activities of the project. This subsection focuses on four main aspects of the model, that are: messages, behavior of participants, protocol sessions, and the security goal. The complete model is given in Appendix C.

**Messages** We model messages, their structure, encapsulation, message encoding and fields by using ASLan++ function symbols and constants as follows:

```
entity Environment {
    symbols {
        % HTTP protocol values
        get, post : method;
        code_30x, code_200 : code;
        uri_sp, uri_i : uri;
        c, sp, idp : agent;
        id : int;
    }%
[...]
```
Symbols for function \texttt{httpRequest}, \texttt{httpResponse}, \texttt{authnRequest} and \texttt{authnResponse} model the structure of a protocol message. \texttt{httpBinding} and \texttt{postBinding} represent messages encoding. Constants \texttt{get}, \texttt{post}, \texttt{code_200} and \texttt{code_30x} model the HTTP GET method, the HTTP POST method, HTTP 200 response code, and the HTTP 30x-family response codes.

By composing together those symbols we express protocol message encapsulation. For example, let us consider the following composition:

\begin{verbatim}
httpResponse(code_30x, idp, httpBinding(authnRequest(sp, idp, id), uri), nil_http_element)
\end{verbatim}

This fragment shows an authentication request \texttt{authnRequest(sp, idp, id)} encoded using HTTP binding and transported over an HTTP message.

\textbf{Client entity} The Client entity needs four parameters called \texttt{Actor}, \texttt{SP}, \texttt{IdP} and URI representing, respectively, the agent that will play the entity of Client, the service provider, the identity provider, and the resource that C wants to access.
As we said in Section 3.1, we defined C as a web browser guided by a user. In our model we considered a standard browser that is not aware of web protocols transmitted over HTTP. We modeled this behavior by using ASLan++ compound types. Listing 3.5 shows the definition of two compound types \texttt{Areq} and \texttt{ARsp}, respectively, the SAML authentication request and response. The behavior of C is specified in the body section of \texttt{Client} entity as follows:

\begin{verbatim}
body {
  % % C-SP (1)
  [Actor]_\[pk(Actor)] *-\rightarrow* SP : httpRequest (get, C_on_uri:(URI),
                                             nil_http_element, nil_http_element);
  SP *-\rightarrow* [Actor]_\[pk(Actor)] : httpResponse (code_30x, ?IdP,
                                               ?AReq, nil_http_element);
}
\end{verbatim}

A Client sends an HTTP request for accessing a resource URI available at SP, obtaining in return an HTTP 30x redirection. The HTTP response transports the new destination IdP and the authentication request AReq. The ASLan++ fragment above shows also which channel properties are used. Here messages are exchanged over SSL/TLS with server authentication. This is specified by using two features of ASLan++: the bullet notation and pseudonyms. A bullet * preceding and following the symbol \rightarrow defines an authentic and confidential channel (see [5]). Pseudonyms allows entities to address the communicating counter-part by the means of a pseudonym instead by their real identity. The composition of the two and by using pseudonyms only on the client side we model an SSL/TLS with server authentication channel.

Thereafter, the web-browser executes the redirection forwarding AReq to IdP as follows:

\begin{verbatim}
% % C-IDP
Actor *-\rightarrow* IdP : httpRequest (get, IdP, AReq, nil_http_element);
IdP *-\rightarrow* Actor : httpResponse (code_200, nil_agent,
                          nil_http_element, htmlForm(?SP, ?ARsp));
\end{verbatim}

Here we assume that C and IdP communicate using mutually authenticated SSL/TLS channels. The authentication response ARsp is carried in an
HTML form (htmlForm) in the body of the HTTP response. Due to user interaction or the execution of a self-submitting client-side script, the web browser will perform another HTTP request in order to transmit the authentication response to the service provider. This is modeled as follows:

 ServiceProvider entity The ServiceProvider entity has four parameters. They are Actor, IdP, C and URI representing the agent that will play the entity of ServiceProvider, the identity provider, the client, and the resource it hosts.

The ServiceProvider entity begins the SAML SSO authentication protocol after receiving a request for URI. In the following ASLan++ fragment, the service provider generates a fresh ID and issues an authentication request:

IdentityProvider entity The IdentityProvider entity has four parameters: Actor, C, SP and TrustedSPs representing, respectively, the agent that will play the identity provider entity, the client, the service provider, and a set of trusted service providers.
IdentityProvider (Actor, C, SP: agent, TrustedSPs: agent set) {
  body {? ...}
  C -->* Actor : httpRequest (get, Actor, httpBinding(
    authnRequest (?SP, Actor, ?ID), ?URI),
    nil_http_element);
}

Once the identity provider receives an authentication request from C, it issues an authentication response on condition that the SP that issued the request is in the set of trusted service providers:

if (TrustedSPs->contains(Sp)) {
  Actor -->* C : httpResponse (code_200, nil_agent, nil_http_element,
    htmlForm(Sp, postBinding(
      signedAuthnResponse (inv(pk(Actor)), Sp, Actor, C, ID), URI)));
}

Sessions and environment The situation we are capturing involves a trusted identity provider, two service provides, one of which is malicious, and a client. We suppose that a user asks for two different resources (e.g. by using two instances of a web browser) being authenticated by the same identity provider. We specified this scenario in the Environment entity that instantiates two protocol executions. A single protocol execution is defined in the Session entity that in turns instantiates participants.

The fragment below shows the body of the Environment and Session entities:

% [...] entity Session (C, IdP, SP: agent, TrustedSPs : agent set, URI : % uri) {
  % [...] entity Client (Actor, SP, IdP: agent, URI : uri) {
    % [...] entity IdentityProvider (Actor, C, SP: agent, TrustedSPs: agent set) {
      % [...] entity ServiceProvider (Actor, IdP, C: agent, URI : uri) {
        % [...] body {
          % a new session is built
          new Client(C, SP, IdP, URI); new IdentityProvider(IdP, C, SP, TrustedSPs);
          new ServiceProvider(SP, IdP, C, URI);
        } goals C_on_uri(_, _) C --> SP;
        % Sessions
        % new Session(c, idp, i, TrustedSPs, uri_i); % honest C talking to i(SP)
        % new Session(c, idp, sp, TrustedSPs, uri_sp); % honest IdP and SP
      }
    }
  }
}
Security goals  In this model we define an authentication property by using ASLan++ primitives. The property aims to verify whether SP authenticates C on URI and it is specified with the following statement \( \text{C_on_uri:(_C) } \rightarrow SP \) placed in the goals section of the entity Session. C_on_uri is an unique identifier for the property that is used by all participants involved according to the ASLan++ language specification [5].
4 OpenID

4.1 Application Scenario Description

OpenID Authentication 2.0 (hereafter OpenID) is an open and user-centric Web browser based Single Sign-On protocol. It provides a way to authenticate a user asking her to prove that she controls an identifier [9]. OpenID is decentralized in the sense that it does not require relying parties (RPs) and OpenID identity providers (OPs) to have a pre-established relationship. It also does not rely on existing infrastructure on which a central authority approves or registers relying parties or OpenID providers. The OpenID Authentication 2.0 specification [9] describes an authentication protocol as well as two optional request-response sub-protocols called association session protocol and signature verification protocol. It also prescribes how messages are transported over HTTP messages defining two communication types: direct communication and indirect communication. The former is established between service and identity providers, the latter involves the user agent as intermediary.

4.1.1 Authentication Protocol

The protocol is initiated by C who access a resource URI at RP providing RP an identifier that C has to prove to control. The identifier is used by RP to identify which OP C uses for authentication. Then C is redirected to OP together with an authentication request. Once C proves to control the identifier, OP issues and signs a positive assertion and redirects C back to the RP transporting the response. RP checks the validity of the signature and it lets C access resources available at its site. The manner in which C is challenged is out of the scope of the protocol specification.

4.1.2 Association Session Protocol

The association session protocol establishes a shared secret $K_{HMAC}$ between RP and OP used to sign and verify authentication responses. RP initializes this sub-protocol by sending an association session request to OP right after RP discovers which OP C uses to authenticate. OP returns to RP a shared secret together with a value $h$ called handle used as a key to refer to associations. OpenID specifies only two ways to transmit $K_{HMAC}$, that are: No-Encryption association sessions and Diffie-Hellman (D-H) association session. When No-Encryption is used the OP sends a response with $K_{HMAC}$ in plain-text, whereas when D-H association is deployed a D-H shared
key is calculated in order to encrypt \( K_{\text{HMAC}} \). No-Encryption is used only over a secure transport layer.

**Diffie-Hellman session association** RP and OP optionally run a D-H key exchange protocol in order to generate a shared key that is used to protect the shared secret. First, RP computes the private key \( x_a \) and sends to the OP a modulus \( p \), a generator \( g \) and \( g^{x_a} \mod p \). OP chooses its private key \( x_b \), calculates the D-H shared key \( g^{x_a x_b} \mod p \), and chooses the shared secret \( K_{\text{HMAC}} \). Then it replies back to RP sending \( g^{x_b} \mod p \), a session handle \( h \), and the shared secret protected by the D-H shared key \( K_{\text{HMAC}} \otimes H(g^{x_a x_b} \mod p) \), where \( H \) is either a SHA1 or SHA256 hash function. \( K_{\text{HMAC}} \) is used both by OP to sign authentication responses and by RP to verify them.

### 4.1.3 Signature Verification Protocol

The signature verification protocol verifies the signature of authentication responses. If RP hasn’t yet established an association with OP, it sends back to OP the authentication response. Upon verifying the signature, OP answers RP with a positive or negative response, accordingly.

### 4.1.4 OpenID Authentication Protocol with Session Association

Figure 8 depicts the authentication protocol flow used in combination with the D-H session association. In step A1, C sends to RP his identifier \( \text{identifier}(C) \). RP identifies OP using C’s identifier (this procedure is not considered here, we just assume that RP has a look-up table) and then initiates the D-H association session protocol. At the end of its execution, RP receives an handle for the association \( h \) and a shared secret \( K_{\text{HMAC}} \) (step S2); then RP issues an authentication request in step A2. C, acting as intermediary, redirects the request to OP (step A3), which challenges C and issues an assertion within an authentication response accordingly. The information sent to RP is signed calculating an HMAC over OP, C, RP, a nonce \( n \) and the handle \( h \) (step A4). In step A5, C delivers the response to RP. If RP accepts the response, then it will send back to C a resource.

**Secure transport layer** The OpenID authentication protocol does not prescribe the use of secure transport layer when exchanging authentication request and responses.

4.2 Security Goals

**Authentication** OpenID has to guarantee *mutual authentication* between C and RP. Mutual authentication is defined as the conjunction of the following two authentication goals:

- *Client authentication*: a relying party RP authenticates a user C on a resource available at the address URI.
- *Relying party authentication*: a user C authenticates a relying party RP on the resource RP sends to C.

**Confidentiality** OpenID comprises three instances of confidentiality properties:

- *Confidentiality of resource*: the resource provided by RP to C must be kept secret between the two of them.
• **Confidentiality of authentication data**: i.e., the authentication requests and assertions must be kept secret between protocol participants.

**Authorization** When an OpenID run ends, RP establishes a local session \( \text{lid} \) with C. The authorization goal is defined as follows: whenever a user C accesses a resource \( r \) available at RP’s site using a local session, it happened in the past that C’s OP issued an assertion \( a \) for C to access RP, \( a \) caused \( \text{lid} \) and \( \text{lid} \) is a valid session cookie of C.

### 4.3 Potential Vulnerabilities

Vulnerabilities listed below come from two different sources: OWASP vulnerabilities categories [16] and **Common Weaknesses Enumeration** (CWE) archive [8]. The classification adheres to OWASP categories except for the cross-site request forgery vulnerability while definitions are taken from both OWASP and the CWE archive. Also, the CWE archive is used to provide more details on how vulnerabilities may occur.

**Input validation vulnerability** An input validation vulnerability arises when a web application does not validate or incorrectly validate input. An attacker is able to craft the input in a way that is not expected by the application. This may result in altered control flow, arbitrary control of a resource or arbitrary code execution (see CWE-20).

Input validation vulnerability might affect an OpenID implementation in several ways. For example, query string values can be shown inside an HTML pages or inserted as HTTP location headers without being validated. Attackers exploit these vulnerabilities by mounting Cross-Site Scripting, SQL Injection, OS Command Injection, and XPath Injection attacks. Examples of weaknesses causing input validation vulnerability are:

- **Improper Neutralization of Special Elements in Output Used by a Downstream Component**: it occurs when a web application constructs all or part of a command, data structure, or record using externally-influenced input, but it does not neutralize or incorrectly neutralize special elements that could modify how it is parsed or interpreted when it is sent to a backend component. The lack of improper neutralization leads to injection problems as SQL injections (CWE-74).

- **Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)** it occurs when a software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users (CWE-79).
Cross-site request forgery vulnerability  A cross-site request forgery vulnerability occurs when a web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.

When a web application receives a request from a client without any mechanism for verifying that it was intentionally sent, then it might be possible for an attacker to trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc. and can result in exposure of data or unintended code execution.

Weaknesses causing Cross-site request forgery vulnerability are:

- **Insufficient Verification of Data Authenticity**: it occurs when a software does not sufficiently verify the origin or authenticity of data, in a way that causes it to accept invalid data (CWE-345).

- **External Control of Critical State Data**: it occurs when a web application stores security-critical state information about its users, or the software itself, in a location that is accessible to unauthorized actors. If an attacker can modify the state information without detection, then it could be used to perform unauthorized actions or access unexpected resources, since the application programmer does not expect that the state can be changed. When this state information is used to control security or determine resource usage, then it may create a vulnerability (CWE-642).

Session Management vulnerability  Session Management vulnerability groups together weaknesses and bugs that affect the way a service provider manages user sessions. They may occur both at architecture and at implementation level.

- **Improper authentication**: it occurs when a user claims to have a given identity (during the execution of an authentication protocol or by providing a cookie) and service implementation fails in proving that the claim is correct (CWE-287).

- **Insufficiently protected credentials**: user credentials can be transmitted or stored by using an insufficiently secured method that leads to unauthorized interception or retrieval. For instance passwords may be stored as plaintext or schemes for recovering passwords are based on one security question. (CWE-522).
• External Control of Assumed-Immutable Web Parameter: it occurs when a web application does not sufficiently verify inputs that are assumed to be immutable but are actually externally controllable, such as hidden form fields, parameters, cookies, or URLs. Web applications can mistakenly make the assumption that data passed is not susceptible to tampering. Improper validation of data that are user-controllable can lead to the application processing incorrect, and often malicious, input. For example, custom cookies commonly store session data or persistent data across sessions. This kind of session data is normally involved in security related decisions on the server side, such as user authentication and access control. Thus, the cookies might contain sensitive data such as user credentials and privileges. This is a dangerous practice, as it can often lead to improper reliance on the value of the client-provided cookie by the server side application. This weakness is required in order to mount session fixation attacks (CWE-472).

Insecure transport vulnerability Insecure transport vulnerability may occur when a program transmits sensitive or security-critical data in the clear in a communication channel that can be sniffed by unauthorized actors (CWE-319). Web applications may not set the Secure attribute (See [13]) for sensitive cookies in HTTPS sessions, which could cause the user agent to send those cookies in plaintext over an HTTP session (CWE-614).

Insecure transport vulnerability may also be caused by a wrong use of the secure transport layer. Figure 9 shows an example of misuse in which RP uses two end-points $RP_{IEP}$ and $RP_{SEP}$ for consuming assertions. The former is an insecure end-point while the latter is a secure one. In step M1, OP redirects an OpenID message to RP through C using the end-point $RP_{IEP}$. Instead of consuming M2, RP redirects C toward the secure end-point $RP_{SEP}$ (steps M4 and M5) where the assertion will be consumed. This results in first transmitting M2 and M3 as clear text and then in re-sending the very same OpenID message over a secure channel.

4.4 Potential Attacks

Cross-Site Request Forgery attack In general, a CSRF attack aims to execute unauthorized commands at the RP’s site from C (the victim). CSRF attacks can be mounted as follows. An attacker prepares a web page containing two hidden iframes. The first one causes the user to visit OP in order to be silently logged in at a targeted RP. The second iframe contains a URL encoded command that the user’s browser will execute at the RP’s site.
Login Cross-Site Request Forgery attack

Login Cross-Site Request Forgery is a variant of CSRF in which an attacker uses the victim’s browser to forge a cross-site request to the honest OP login URL, supplying the attacker’s user name and password. A vulnerable OP will execute this request and log the victim into the site as the attacker [6].

The attack is mounted by an attacker as follows. First, he begins the authentication process with RP and OP by providing his own credentials to OP. Then he interrupts the protocol when OP redirects the attacker to the RP end-point. At the same time the attacker is acting as a malicious web-site with a victim C and, instead of executing the redirection, he redirects C to the RP together with the positive authentication assertion. When RP consumes the assertion, it stores in the victim’s browser session cookies referring to the OpenID session the attacker initiated.

The vulnerability is due to a lack of mechanisms either at specification or at implementation level that bind an OpenID session to user’s browser. However, the OpenID specification states that the return address OP uses to address responses to RP may be used by RP as a mechanism for attaching context about the authentication request to the authentication response [9].
CSRF attacks violate security goals such as the authentication goals between C and RP. Those attacks mainly rely on the presence of Cross-site Request Forgery vulnerabilities.
5 OAuth 2.0

OAuth 2.0 (hereafter, OAuth2) is an authorization protocol aimed at granting to service A access to a protected resource (e.g. hosted on a service B) on behalf of a user, without requesting the provisioning of the user’s credentials to the service A [12].

5.1 Application Scenario Description

5.1.1 Overview

On the Web, users access various online services, such as photo sharing and e-banking services. In general, each of these services are accessed and used once the user is authenticated and, thus, recognized by the service. Whenever each service is used individually, the user simply connects to the service, authenticates himself and then uses the service.

Problems arise when services can act in a cooperative way, i.e. when a service A requires to access to a user resource hosted on another service B on behalf of the user. A trivial solution would be the provisioning of user’s credentials for B service to A. However, this choice has many drawbacks (e.g. potentially insecure, difficulty in revoking grants, ...) and it is far from being a suitable solution, in particular if the user often needs to delegate the use of a protected resource to other services.

OAuth2 introduces an authorization layer that separates the roles of the user and the service aiming at accessing the user’s protected resource, thus avoiding the need of providing of user’s credentials to services. In particular, the protocol carries out the authorization process by orchestrating an interaction between the user and the service.

OAuth2 is an open and HTTP-based protocol defined as an extension of OAuth 1.0; however, it is not backwards compatible with the first version (see [12] for further details).

The protocol involves four roles (see Figure 10):

1. the Resource Owner (RO), which corresponds to an entity capable of granting access to a protected resource (i.e. the user),

2. the Resource Server (RS), which hosts the protected resource,

3. the client (C), namely an application that aims to access the protected resource on behalf of RO
4. the Authorization Server (AS), which issues an authorization grant to C, after successfully authenticating the RO and obtaining authorization.

![Diagram of OAuth2 flow]

Figure 10: Abstract flow of OAuth2

The AS is divided into two endpoints: (a) the Authorization EndPoint (AS_AEP) which is responsible for the interaction with the RO, for authentication and authorization aims, and the (b) Token EndPoint (AS_TEP), which is responsible for issuing tokens. Beyond active roles, passive roles like User Agent (UA) (i.e. the browser) supporting the interaction between the RO and the AS, and some external servers (XS) can be involved.

Authorization grants are delivered by the AS in form of Access Tokens (AT). The possession of the corresponding AT allows C to access the protected resource, by presenting the AT to the RS. However, this last step is outside the scope of OAuth2, where the execution terminates after the issuing an AT to C.

5.1.2 OAuth2 flows

The issuance of an access token to the client can be carried out through distinct variants of the protocols, called flows. OAuth2 supports four flows. Each flow involves the entities in different ways. The choice among flows depends on which security conditions can be granted by the involved entities. Supported flows are: Authorization Code flow, Implicit flow, Resource Owner flow, and Password flow.
Credentials flow, and Client Credentials flow. Most of the current OAuth2 implementations supports the Authorization Code and Implicit flows, thus we only describe them [12].

Authorization Code Flow The Authorization Code Flow (hereafter, ACF) requires the authentication of C at the AS, thus it is strictly suitable for clients able to keep their credential secret (e.g. applications running on a Web server instead than in a browser). Credentials of a client are a pair \((c_{id}, c_{secr})\) where \(c_{id}\) uniquely identifies C at AS and can be publicly known; \(c_{secr}\) corresponds to a secret shared between AS and C and must be kept secret by the client.

The Authorization Code flow is summed up in Figure 11. Each message has been defined using functional symbols to distinguish between different messages and isolate each component of a message. In particular, functional symbols \(H_{Req}\) and \(H_{Resp}\) indicate HTTP Request and HTTP Response, while \(h_{el}\) indicates a single element of an HTTP message.

Figure 11: OAuth2 Authorization Code flow

The Authorization Code flow is defined as follows:

- **(steps 1-2):** the client starts the flow by directing the Resource Owner’s user-agent to the authorization endpoint through an HTTP Request (\(h_{Req}\)). The client must include its client identifier \((c_{id})\), the \(response\_type=\text{“code”}\), which uniquely marks the flow as the authentication code one, and a redirection URI to which the AS_AEP will send the user-agent back once access is granted (or denied). Optional fields can contain information on the scope of the authorization (SC) and state information (ST) between C and AS. \(g\) indicates the method GET of the HTTP Request.
• (step 3): the AS_AEP authenticates the RO (via the user-agent) and establishes whether the Resource Owner grants or denies the client’s access request.

• (steps 4-5): if RO grants access to C, the AS redirects the UA back to the client to the redirection URI provided earlier via an HTTP Response message (hResp). The redirection URI includes an authorization code and any ST information provided by the client earlier. Thus, the UA redirects the received message to C.

• (step 6): once C requires to access the protected resource, it contacts the AS_TEP, providing the authorization code and its credentials ($c_{id}$ and $c_{secr}$).

• (step 7): the AS_TEP authenticates C and checks the authorization code. If both verifications are correct, it issues an Access Token (AT) directly back to C, together with, optionally, a Refresh Token (RT).

C can then access the protected resource by contacting directly the Resource Server (RS) and using its AT.

An AT can have a limited validity duration. Once the access token has expired, it is no more accepted by the RS. In this case, if a RT has been provided to C, C repeats the protocol from step 6, by providing the RT instead of the authorization code, thus receiving a new and valid AT. In case C has not received a RT and the AT expires, the whole protocol run must be re-executed from the beginning, involving the RO.

Implicit Flow The Implicit Flow (IF) is suitable for clients that are not able to keep their credentials secret (e.g. applications running inside a browser). In this case, each client still has a unique identifier $c_{id}$ but no secret $c_{secr}$. This flow does not require a full authentication of C. The IF is summarized in Figure 12.

• (steps 1-2): like in the ACF, C redirects the UA to the AS_AEP in order to obtain authorization from the RO. C provides its client identifier ($c_{id}$), the redirection URI and sets the $resp\_type=token$. $g$ indicates the GET method. Scope and state fields can be provided optionally. As mentioned, no client secret is provided in this case.

• (step 3): the RO, redirected through the UA to AS_AEP, authenticates itself and grants access to C.
• (step 4): the AS_AEP sends an AT directly back to the UA, contained in a URI fragment (#) of the redirection URI.

• (steps 5-6): the UA contacts an external server which provides back a script that, once executed on the UA, allows to retrieve the AT from the fragment. The URI fragment with the AT must be kept on the UA and not delivered to the external server.

• (step 7): the UA executes the script and extracts the token.

• (step 8): the UA delivers the AT to C.

5.1.3 Protocol assumptions

The protocol is expected to run on actual systems under the following assumptions:

• Secure channels. Messages with the AS (both AS_AEP and AS_TEP) must be exchanged on secure SSL/TLS channels.

• Tokens. Tokens and authorization codes are exchanged between C and AS in order to obtain authorization; therefore, C and AS must store them in a confidential and tamper-proof manner.

• Correlation between codes and tokens In ACF, the authorization method of OAuth2 strongly relies on the possibility to correlate value of the authorization code (e.g. exchanged in the early phases of the protocol) with access token issued at the end of the protocol run. Besides, same correlations must be kept and checked between refresh tokens and access tokens by the AS.
5.2 Security Goals

OAuth2 security goals are distinguished between the Authorization Code Flow (ACF) and Implicit Flow (IF).

**Authentication**  Authentication regards Client (C), and Resource Owner (RO).

- *Client authentication in ACF.* The Authorization Server EndPoint (AS_AEP) authenticates the Client (C) through its full credentials (client identifier and client secret) and they agree on the values of the exchanged values of authorization code.

- *Client authentication in IF.* AS authenticates C on its partial credentials (on the client identifier only) and they agree on the value of the exchanged access token.

- *Resource Owner’s authentication.* The RO must be authenticated by the AS_AEP during the protocol run through RO’s credentials.

**Authorization**  OAuth authorizes C to access some protected resources, owned by RO, by providing an access token to C. It is fundamental to verify that if C is granted access to a protected resource through an access token, then an Authorization Server has previously issued that token to C, after having received the access grant from RO.

**Confidentiality**  Confidentiality is related to different values:

- *Tokens and authorization codes.* Authorization code must be kept secret between AS, UA and C, while access and refresh token must be kept secret between the AS and C during the protocol run (only ACF).

- *Client credentials.* In ACF, Credentials must be kept confidential by the client and shared exclusively with the AS on a secure channel.

- *Resource Owner credentials.* They must be shared only with the AS (in both flows).

**Integrity**  Access tokens, refresh tokens and authorization codes delivered by the Authorization Server endpoints (i.e. AS_AEP and AS_TEP) must reach the destination client without being tampered. The integrity of code and tokens is related with authentication goals, since the violation of integrity would take to the impossibility to authenticate participants.
5.3 Potential Vulnerabilities

Actual systems implementing OAuth2 can suffer from the following vulnerabilities:

Session Management vulnerabilities An insecure OAuth2 implementation might allow an intruder to steal the authentication code in an ACF and use it on the AS_TEP in order to obtain the access token instead of the honest client. Moreover, weak algorithms (e.g., using surjective functions with low entropy and/or with predictable distribution) for calculating access tokens, refresh tokens and authorization codes can allow an intruder to guess and generate such values. In this way, the intruder can use the spoofed tokens/codes in order to mislead the Authorization Server or the Resource Server.

Input Validation Lack of input validation in the actual implementation could lead to severe vulnerabilities. In particular, the endpoints of the Authorization Server should validate every input coming from the client and the Resource Owner in order to avoid stored XSS attacks (see [22]).

Path vulnerability Since the protected resource is identified by a URI, such element, if not properly formed, can provide information on the Resource Server file system, thus providing unwanted information for a path traversal attack.

Password Management vulnerabilities Credential information regarding C (i.e., cid and csecr) and RO (i.e., user credentials) are both negotiated with the AS and stored on it. Besides, RO’s and C’s credentials can also be stored on the UA during the redirection steps. If AS and UA do not securely manage the storage of credentials, this information can be lost or stolen. Moreover, if C chooses a guessable secret (csecr) and RO has a too simple password, then the AS can be the target of password guessing attacks. Although these aspects are important from a security point of view, they should be considered out of the project scope.

5.4 Potential Attacks

5.4.1 Violating authentication

Credential guessing attacks Aimed at guessing values of access tokens, authorization codes, refresh tokens, RO passwords and client secret. It is
related with the AS implementation and the algorithms used for generating tokens and codes but they are out of scope in the project.

Credential guessing attacks may be performed by exploiting password management vulnerabilities.

**XSS attack**  An intruder that is able to perform a session fixation attack and, thus, accessing the resource, can tamper the resource by injecting malicious code so that the honest client that successively accesses the resource can inadvertently execute the code. XSS attacks can also be achieved by tampering with the redirection URI or the scope or state fields in the client requests.

XSS attacks may be performed by exploiting input validation vulnerability.

5.4.2 Bypassing authorization

**Session Fixation attack**  It leverages the authorization code grant type, by tricking an end-user to authorize access to a legitimate client, but to a client account under the control of the attacker. It can be avoided by verifying that the value of the redirection URI is the same on both authorization code and access token requests.

Session Fixation attacks may be performed by exploiting input validation and session management vulnerabilities.

**Cross-site request forgery (XSRF) attack**  A client who possesses an access token can be mislead to use it on the resource in order to perform unwanted actions on it. In this case, a malicious user interacting with the client can force it to act by exploiting vulnerabilities in the client implementation. In particular, a malicious user can force the client to use the access token to perform a given action (e.g. accessing the resource on the Resource Server) without the client itself being aware of this.

XSRF attacks may be performed by exploiting path validation and session management vulnerabilities.
6 Pervasive Retail

6.1 Application Scenario Description

6.1.1 Overview

The Pervasive Retail application scenario is contributed by SAP. It contains a new on-demand marketing management solution to create interactivity between consumers, and retailers/product providers through mobile phones to provide product and promotion information to consumers, influence consumers at the point of decision, increase consumer traffic to retailers and in-store sales. In this context, products are produced by product providers, sold by retailers, and bought by consumers. For example, Alice could be a consumer, a shampoo is a product, Casino is a retailer, and L’Oréal is product provider. Casino sells L’Oréal’s products such as shampoo to consumers like Alice.

For the consumer, a Personal Shopping Assistant which is a software application is provided that delivers rich product information and special offers in real time through mobile phones. For the Retailer and the Product Provider, a Marketing Management Platform is provided that enables real time interaction with the consumer at the point of decision to inform and influence consumer behavior, drive consumer traffic and increase average spend. Retailers and Product Providers can interact with consumers in the field, one-to-one, in real-time with a very high level of personalization for maximum impact.

6.1.2 Scenes Description

In this application scenario, two scenes will be considered: the Individual Offer and Coupon. They are depicted in Figure 13-Figure 14 and detailed hereafter.

Individual Offer Scene

As depicted in Figure 13, in the first scene, retailer provides an individual offer to a consumer based on her loyalty program status (the loyalty program is run by the retailer) and the product concerned. The story line is as follows:

- A consumer is doing shopping in a store. She is interested in a specific product. She scans the barcode of the product using her mobile phone, and sends the picture to the Pervasive Retail Platform.
• The Pervasive Retail Platform receives the picture, identifies the barcode, determines the product by looking up in product catalog, and collects all the information about this product.

• The Pervasive Retail Platform also identifies the consumer. By looking up in consumer database, the Pervasive Retail Platform collects all the information about the consumer including her loyalty program status, shopping history, and shopping preference.

• Based on the product information and consumer information, by looking up in promotion database, the Pervasive Retail Platform selects some offers for the consumer, and sends them to the consumer.

• After receiving the offers provided by the Pervasive Retail Platform, the consumer can decide to purchase the related products or not. If yes, she goes to the cashier of the retailer to buy the products according to the offer received.

• After the products are sold, the Pervasive Retail Platform collects the information about purchasing behavior, and sends related information to Product Provider.

• Based on all the information received, the Product Provider may optimize the advertisement campaign. This action may not be performed in real-time unlike the other actions.
Figure 14: Scene 2: Coupon

Coupon Scene

As depicted in Figure 14, in the second scene, Product Provider provides a coupon to consumer, and the Pervasive Retail Platform provides rich product information including direction to stores. The story line is as follows:

- The consumer is in a street of a city, and sees a printed advertisement about a specific product. She is interested in the product. She takes a photo of the advertisement using her mobile phone, and sends it to the Pervasive Retail Platform. In order to know some stores nearby selling the specific product, the consumer may also send her position information to the Pervasive Retail Platform.

- The Pervasive Retail Platform receives the photo, identifies the product, and by looking up in product catalog collects all the information about this product.

- The Pervasive Retail Platform also identifies the consumer. By looking up in consumer database, the Pervasive Retail Platform collects all the information about the consumer including her loyalty program status, shopping history, and shopping preference.

- The Pervasive Retail Platform also identifies the Product Provider of the specific product, and sends the related information about the consumer and the product to Product Provider.
• After receiving all these information sent by the Pervasive Retail Platform, the Product Provider evaluates the status of the consumer, and may decide to give her a coupon for the specific product. In this case, the Product Provider sends the coupon to the Pervasive Retail Platform.

• At the same time, the Pervasive Retail Platform collects information, such as reviews, about the specific product from the web, and sends the information to the consumer. If the consumer has also sent her location information, the Pervasive Retail Platform also sends to the consumer the information about stores nearby that sell the product. The store information includes the position of the store, the direction to the store, and the related price information. In case a coupon is received from Product Provider, the Pervasive Retail Platform registers the coupon, and sends it to the consumer.

• After receiving all the information about the product, the stores and the coupon, the consumer can decide to buy the product or not. If yes, the consumer goes to a store to buy the product showing the coupon received. The store interacts with the Pervasive Retail Platform to process the coupon.

• After the products are sold, the Pervasive Retail Platform collects the information about purchasing behavior, and sends related information to Product Provider.

• Based on all the information received, the Product Provider may optimize the advertisement campaign. This action may not be performed in real-time as the other actions.

6.1.3 Architecture Description

A high level system architecture of the Pervasive Retail application scenario is depicted in Figure 15, in which multiple retailers, Product Providers, and of course consumers are served by one single Pervasive Retail Platform, which could be hosted by SAP. The consumers interact with the platform using Consumer Clients. The retailers and Product Providers interact with the platform using Retailer Clients and Product Provider Clients respectively, through which the main function provided is analytics of the selling data. Both retailers and Product Providers have their own backend system, e.g., SAP CRM (Customer Relationship Management) system for managing consumer profiles. The Pervasive Retail Platform interacts with these backend
system to obtain information about products, consumers, and promotions, etc.

Figure 15: High Level System Architecture

Figure 16 provides a more detailed “zoomed in” view of the architecture. In the Retailer Backend System, information about product catalog, product price, product availability, and consumer profile are stored in a database.

In the Pervasive Retail Platform, there is also a local database, in which information about product catalog, product price, and consumer profile are stored. The product price stored in Pervasive Retail Platform is basically a cache of the corresponding information in Retailer Backend System. Whenever product price information is obtained from the Retailer Backend System, it is stored in the local database with an expiration time field whose value represents a time span and decreases when time elapses. When the expiration time field decreases to zero, the information is removed from the local database. The product catalog stored in Pervasive Retail Platform is a
replication of the information stored in Retailer Backend System. There is a module named Product Catalogue Replication dealing with the replication. This operation could be triggered in various ways. The product availability information is changing rapidly, so there’s no local copy or cache in the Pervasive Retail Platform. Whenever the information is needed, it is obtained from the Retailer Backend System.

Inside the Pervasive Retail Platform, there are modules IDM (Identity Management) dealing with identity management, Consumer Management managing the consumer profile and shopping preferences, and Offer Management responding to consumer request asking for an offer.

Figure 16: Detailed System Architecture

A sequence diagram in Figure 17 gives an example of the various modules interacting with each other. A consumer requests an offer by invoking an interface `getDeals()` provided by the Offer Management module. The Offer Management module invokes interface `identifyUser()` provided by IDM to obtain user information, invokes `getBestDeals()` provided by itself, invokes interface `getProductInfo()` provided by the product catalog to ob-
tain product information, invokes `getProductAvailability()` provided by the Retailer Backend System to obtain product availability information, invokes `getProductPrices()` to obtain product price information, responds to the consumer with the best offer based on all these information, and caches the product information in local database.

### 6.2 Security Goals

At the security protocol level, the following security goals are expected in the Pervasive Retail Platform:

- **Mutual Authentication.** The Pervasive Retail Platform shall have mutual authentication with Consumer Client, Retailer Client, Product Provider Client, Retailer Backend System, and Product Provider Backend System. During message passing process, channels with confidentiality and integrity properties are used.

- **Secrecy.** In the individual offer scene, the offer chosen by the Pervasive Retail Platform shall be a secret between the platform and the consumer, unless it is exposed by the consumer to the retailer. In the coupon scene, the coupon made by Product Provider to the consumer through the Pervasive Retail Platform shall be also a secret, unless it is exposed by the consumer to the retailer.
• Non-repudiation. In the individual offer scene, the retailer shall not deny the offer made for the consumer. There is another case of non-repudiation in the Pervasive Retail application scenario. When consumer receives a coupon from Product Provider, Product Provider cannot deny it has issued the coupon. In addition to this, since there is a trust relationship, e.g. established by some agreement, between Product Provider and Retailer, Retailer will accept the coupon presented by the consumer.

At the business process level, the following security goals are expected in the Pervasive Retail Platform:

• Authorization. All the data access shall be controlled by authorization. For example, in both scenes of the application scenario, information about purchasing behavior are collected by Pervasive Retail Platform and sent to Product Provider. These purchasing behavior information could be, e.g., the relationship between the purchasing decision and the coupon, or the effect of promotions provided by the Product Provider. At the same time, the exact selling price information in purchasing behavior is a secrecy of Retailer, and shall not be sent and seen by Product Provider.

• Privacy. In the coupon scene, the consumer can decide whether to expose his/her location information to the Pervasive Retail Platform and Product Provider.

• Need-to-Know. In the coupon scene, consumer identity information such as name and gender is not necessary for the Product Provider to make a decision about coupon, so it shall not be sent to and known by the Product Provider.

6.3 Potential Vulnerabilities

In this section, we list some potential vulnerabilities in this application scenario. In this deliverable, we list those vulnerabilities that are most interesting for SAP for the time being. At the same time, they are all related to OWASP Top 10 security risks [15]. During the course of the project, we may update the list based on SAP’s business interests and research results obtained in the project.
Session Management Vulnerability  Authentication mechanisms rely on sessions to differentiate an agent from another one and to have a degree of trust in the identity of another agent. When there are vulnerabilities or flaws in session management, authentication could be affected.

Input Validation Vulnerability  Pervasive retail is a web application. Thus it is assumed that attackers have a full control of the inputs, and that the application could be vulnerable to all kinds of data injection or manipulation. This is the reason why all data must always be validated before being used by the system.

Path Vulnerability  A lot of files can remain on the server and they might be available for each Internet user and specially for search engine robots. This vulnerability can lead to serious leaks of information.

6.4 Potential Attacks

In this section, we list some potential attacks in this application scenario. In this deliverable, we list those attacks that are most interesting for SAP for the time being. At the same time, they are all related to OWASP Top 10 risks. During the course of the project, we may update the list based on SAP’s business interests and research results obtained in the project.

Spoofing attacks such as Man-In-The-Middle  The man-in-the middle attack intercepts a communication between two client and server. This attack exploits vulnerabilities in session management.

This attack could happen between various clients and the Pervasive Retail Platform in this application scenario. This may break the mutual authentication and secrecy goals.

Path Traversal Attack  This category of attacks tries to access files or directories that are not intended to be accessed. This attack works on applications that take user input and use it in a “path” that is used to access a file system. If the attacker includes special characters that modify the meaning of the path, the application will misbehave and may allow the attacker to access unauthorized resources. This attack exploits input validation vulnerabilities and path vulnerabilities.

The various information stored in the Pervasive Retail Platform could be target of this attack. This attack may break the secrecy and authorization goals.
**Cross Site Request Forgery**  CSRF is an attack that leads an end user to execute unwanted actions on a web application in which he/she is currently authenticated. With a little help of social engineering (like sending a link via email/chat), an attacker may lead the users of a web application to execute actions of the attacker’s choosing. A successful CSRF exploit can compromise end user data and operation in case of normal user. This attack exploits session management vulnerabilities.

The interaction between Consumer and Pervasive Retail Platform could be target of this attack. This attack may break the authentication goals.

**Cross Site Scripting**  Cross-Site Scripting (XSS) attacks are a type of injection problem, in which malicious scripts are injected into the 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. This attack exploits input validation vulnerability.

There are various clients interacting with the Pervasive Retail Platform. XSS attacks could happen in all these interactions. Various security goals such as secrecy and authorization could be broken by this attack.
7 Infobase Document Repository

7.1 Application Scenario Description

7.1.1 Overview

The Infobase Document Repository (IDR) implements a Document Management System that allows for the secure management and sharing of any documents or data files using only a web browser. It is provided by Siemens to offer a collaboration platform for joint projects that involve external and internal partners.

The repository mechanism supports the web-based administration of text and binary files of any kind, e.g., Word documents, Excel tables, and even executables, in a hierarchical storage structure. The following characteristics can be noted:

- Up- and download of entire directory trees as zip archives
- Version management
- File locking for team-oriented editing
- Cut/copy/paste mechanisms for files and directory trees via a clipboard
- Symbolic links
- Finely-granular access control, in which separate access rights can be allocated to every repository object (users, groups, company rights).

The main features of the system are as follows:

Repositories. The IDR system contains a set of workspaces called *document repositories*. Each repository has an owner, usually a user group or a company who is responsible for the content. An owner can own several repositories.

Folders. A single repository can be considered as a shared file system, where the documents are organized into hierarchies of folders.

Links. Links can be created to any document or folder and placed in any other folder.
Access Permissions. The access to the documents can be restricted through a fine-grained set of permissions, where each action such as view, edit, download, overwrite, or delete can be restricted to arbitrary sets of users or user groups. Permissions can be attached also to folders where the child documents and folders can inherit them. Child documents or folders can overwrite any parent permissions, allowing a flexible access management. Users see only those repositories, directories, and files that they have at least the listing right for.

Full Permission List. A full list of all users with their personal access permissions can be viewed separately for each document by an administrator or by a user who is permitted to access the document.

Document Locks. In order to support project teams that work simultaneously on various tasks, virtual locks are implemented. Documents can be locked by editors for a given period of time, preventing concurrent changes, and unlocked when the work is finished and the last version has been uploaded.

Versioning. The repository implements a simple revision system where each document has a list of its archived versions. Each file can be overwitten by a new version with or without keeping the old version. In both cases, the object ID, which is relevant for direct links to the files, stays unchanged. So links in documents or in web pages leading to certain files in the repository are not damaged by updating files. A link always leads to the newest version of the file. Older versions of the file are stored persistently and can be restored if necessary. Archived versions have the same permission as the relevant current version of a document. The numbering scheme of the version management is quite simple. It starts with version 0 and counts up to 1, 2 and so on. To support the user in the off-line handling of file versions, a version-suffix (_0, _1, _2 etc.) is appended to the file name automatically as a naming proposal for the download.

Repository Tree View. A tree view of a whole repository or any given folder provides a fast access to documents and folders within complex hierarchical folder structures.

Clipboard. A simple clipboard enables useful functions like copy, cut, move and paste for any document or folder within the repository.
Notification. Notification emails can be sent to notify certain users or user groups when specific actions performed on a specific document or folder.

Advanced Actions. To simplify the use of the repository there are different actions like downloading a folder as a zip file and uploading a folder as zip file for complete folder structures.

Properties. Each document and folder has a set of properties like creator, size, created at, last updated or comment. Keywords can be used to categorize and search the repository.

7.1.2 Scenes Description

Figure 18 provides an overview of the IDR system with actors and all possible actions as well as the various supported access methods to the system.

Possible scenarios for using the IDR system and related threats, risks and vulnerabilities are presented in Figure 19. A description of a specific usage scenario of the IDR system is given below.

Infobase Usage Scenario. Let’s imagine an IDR repository with the name “ABC Inc.”. The repository is set up to serve as a share-point as well as a store for different artifacts produced by teams from the cooperation “Siemens AG” and “ABC Inc.”. One user on each side is granted the access right to the repository: Peter Smith (ABC Inc.) and Maggie Lee (Siemens AG). Maggie Lee is at the same time the repository administrator. The following artifacts are assumed to be stored in the repository: contract documents, price lists, confidential offers to mutual clients etc.

A typical scenario could be as follows:

- Peter Smith (PS) logs in through the login page (Figure 20) by providing his user name and password.

- After a successful login, PS is forwarded to the welcome page of the Infobase system as in Figure 21. PS clicks the “Repository” link in the main menu to get access to the repositories available to him as in Figure 22. Now both public repositories, which are visible to all users, as well as the “ABC Inc.” repository are accessible for PS.

- PS clicks the “ABC Inc.” repository link from the list of repositories to open the repository view (Figure 23).
Figure 18: Assets, actions and stakeholders of the IDR system
Figure 19: Possible scenarios, threats and risks associated with the IDR system
• Now PS uploads a new document (*contract.doc*) to the repository using the “Upload Document” mask and clicks the “Upload” button. Upon successful upload, the “ABC Inc.” repository view looks as in Figure 24.

• Now ML as the repository administrator changes the permissions of PS by clicking the key symbol to the right of the document (Figure 24) and un-checking the check boxes for *Move/Rename*, *Delete/Overwrite* and *Add/Restore Version* to the right of the user PS. After saving the new permission settings, PS is expected to be allowed only to download and read the contract document.

![Infobase login page](image)

Figure 20: Infobase login page

### 7.1.3 Implementation Technologies

The following technologies are used for the Infobase Document Repository.

- **Java Servlets**:
  - Server side Java code, which serves the HTTP requests, executed within a servlet container (Tomcat).

- **Java Server Pages (JSP)**:
  - Enables the design and development of dynamic web pages.

- **Java Tag Libraries**:
  - Extensible logical or GUI components for JSP pages.

---

• Struts (Apache Foundation): A widely used web framework that defines many helpful components and design patterns, such as Model-View-Controller.

\[12\] NetBeans: http://www.netbeans.org/
Struts-Layout\textsuperscript{13}: A tag library with rich GUI components, such as Forms, Tables, Tabs, Menus and JavaScript actions.

Hibernate\textsuperscript{14}: Fast and transparent mapping between the object-oriented

\textsuperscript{13}Struts-Layout: \url{http://struts.application-servers.com/}

\textsuperscript{14}Hibernate: \url{http://www.hibernate.org/}
Java models and the relational database tables. Supports class inheritance, aggregations, associations, lazy instance loading, cascading save/update/delete, instance caching, transaction isolation and automatic object locking and versioning. Hibernate provides simple object-oriented query language.

- **Java Database Connectivity**\(^{15}\): Used by Hibernate to communicate with the database.

- **Java Mail API**\(^{16}\): Used to create and send e-mails.

- **Log4J**\(^{17}\): A Logging utility for runtime log messages of different categories, such as debug, info or error. It is capable of creating and managing different files, formats and message listener. Log4J is also capable of sending e-mails notifications in case of the occurrence of certain messages, for example errors.

### 7.1.4 Architecture Description

The application logic forms the central part of the IDR system. The logic is implemented within Strut Actions, which are responsible for the delegation of concrete work to the different services (Figure 25). The session state is encapsulated within context objects that can be accessed in the same way through the whole system. Action Form is a Strut object that maintains the state of a web application. The state of the application is represented by a model based on Beans matched to the database via Hibernate. While meta-data of all repository objects are stored in the database, repository files are stored directly in the file system and are only referenced in the database.

**Servlet, Interceptor Filter and Actions.** The servlet specification defines the contract between the server side executed code and the server runtime environment. The HTTP requests are accepted from the Apache web server, then decrypted and transferred to Tomcat, which serves as the servlet engine. Tomcat chooses the appropriate servlet and calls its `doPost` or `doGet` method. Each request is served in a separated thread. Therefore access to shared resources must be synchronized. The Action Servlet delegates the concrete work to some services and calls a JSP to create the response. The response is encrypted by the Apache server and sent back to the user. LoginInterceptor (login service) and DBInterceptor (persistence service) are used.

\(^{15}\)JDBC: [http://www.oracle.com/technetwork/java/overview-141217.html](http://www.oracle.com/technetwork/java/overview-141217.html)

\(^{16}\)Java Mail API: [http://www.oracle.com/technetwork/java/javamail/](http://www.oracle.com/technetwork/java/javamail/)

\(^{17}\)Log4J: [http://logging.apache.org/log4j/](http://logging.apache.org/log4j/)
Figure 25: Component structure of the system

to wrap each call to a given servlet (Figure 26). Thus the calls are made secure and bound to a given database transaction.

The Struts framework consists of a single servlet, which uses a request processor to dispatch the HTTP request to the appropriate action handler. The action handlers are defined within a configuration file and mapped to special URL paths. Possible interactions between these components are shown in the sequence diagram in Figure 27. The InfobaseProcessor shown in the sequence diagram does not process the incoming requests itself, rather it delegates the final action to other servlets dedicated for processing specific requests.

7.2 Security Goals

Authentication. User authentication is a strong requirement since much of the information available within the Infobase is subject to confidentiality constraints. Moreover, it is important to distinguish four types of users: external users, repository users (Siemens), repository administrators and system administrators (see Figure 18). Authentication is realized with a login and a password for all users and with a PKI login mechanism for a stronger...
and more comfortable authentication for Siemens employees.

**Access Control.** In order to restrict access to the different IDR assets, a mechanism similar to role-based access control model (RBAC) is used. But, the reader should have in mind that the way the word “role” in infosys does not correspond to the way the same word is used in RBAC, and also note that there is an extra concept of “group”. While RBAC associates users to roles to permissions, the infosys model associates users to groups, and both (users or groups) to roles. Roles are associated to permissions, but not in the infobase administration interface, which is used to administer many types of applications. This association (role to permissions) is dependent on the particular application and is determined by the application code itself. A user can belong to one or several groups; groups may contain 0 or more users. In the same way, a user or a group can be associated to one or several roles,
a role may be associated to any number of groups or users. Within infobase, not only the association in one direction are visible (users to groups to roles), but also in the other direction. As a result, the effort needed to determine the user rights for a particular operation/action is significantly reduced.

Based on this dynamic authorization approach, access rights can be configured by an administrator for each repository and role. Furthermore, these rights can be inherited from the repository to a folder down to a single file. On the other hand, it is possible to grant certain groups or users access rights to individual files in a repository tree structure that is otherwise not accessible for this user group. In this special case, the users with this punctual right have to get a direct link to the file because otherwise they would not be able to browse to its location. The access rights to perform all actions on files and folders, such as to view, move, edit, lock/unlock, comment, delete and modify, are thus only granted if they are inherited from an upper level or explicitly given on the actual level.

The following roles are able to define user access rights:

- **IDR Administrator** has administrative access rights for repositories and all users. S/he modifies access rights throughout the system by assigning or removing roles of users and groups, but is not able to give
permissions to access files or directories within repositories, because for that he would need to program the application, which determines the association of roles to permissions.

- **Repository Administrator** has administration rights for a specific repository. He/she can grant access rights throughout the repository by assigning or removing associations for users and groups or the permissions for the roles.

Thus in particular, to define access control, two approaches are used: on the one hand, it is possible to use groups, or it is possible to grant permissions directly to individual users. In a similar way, it is possible to grant permissions to whole directories/subdirectories, or to particular files. (See the Peter Smith story in Section 7.1.2).

**Confidentiality.** Confidentiality is basically enforced by the means of authentication and the RBAC/DAC. Communication between the client (web browser) and the IDR system is encrypted using TLS (HTTPS).

IDR uses these mechanisms at channel level in order to check that any non-authorized entity cannot understand the content of the artifact provided by the repository. On the other hand, to ensure the confidentiality of information is to avoid the possibility of viewing repository artifacts from not authorized users or eavesdroppers. This is connected to authentication mechanisms, because as long as this latter property is guaranteed, an intruder cannot communicate with the repository in any way. This prevents different types of attacks implying confidentiality violation.

**Integrity.** Similar to Confidentiality, IDR uses authentication, versioning, access control mechanisms and permissions as well as the database and file system capabilities and TLS secured channels to support integrity. This is checked, along with authentication, using standard ASLan++ goals.

**Accountability.** Internal repository users (as well as administrators) using PKI have to authenticate themselves with their smart cards. Every user who wants to access to IDR services needs to be registered and its data stored in a proper database. Thus, for every action performed by a user we can prove that this action was really performed by himself. The respective accountability goal can be modeled in ASLan++ combining different features such as LTL properties, Horn clauses and predicates.

This security requirement has been already considered in the context of the AVANTSSAR project and its use case scenarios. Thus it is possible to
find concrete examples of how to write an ASLan++ specification containing the necessary constructs to formalize accountability and in particular the respective goal. For example, in the Digital Contract Signing (DCS) scenario special facts are used in the Signer Entity to take trace of operations performed by honest agents. Thanks to those facts, and the possibility for the Business Portal Entity to store suitable proofs (using a Secure Archiver Service), it is possible to link each operation with the user data and the proof stored, in order to guarantee accountability. In the case an user denies an action, the Business Portal is able to provide evidence that he really performed it (if the user indeed performed the action). For formal details see [4, Section 4.2.4].

7.3 Potential Vulnerabilities

The vulnerabilities of this case study are basically not different from the vulnerabilities of a generic modern web application. Therefore, as for other case studies, all known vulnerabilities apply. Particularly relevant for InfoBase are:

**Authorization.** Exploiting authorization flaws, for example bypassing the RBAC mechanism, would potentially compromise the confidentiality of the data in one or more repositories (depending on the severity of the flaw), as well as the integrity of the stored information.

**Authentication.** Unauthorized gain of authentication credentials can lead to privilege escalation and thus poses a threat to all of the security goals for this case study.

**Error handling.** If escaping errors is not handled correctly, for example after the incorrect upload of a file to the repository, can lead to an insecure system state by escalating privileges.

**Input validation.** As for other use cases, lack of input data validation leads potentially to cross-site scripting and injection attacks. As this may lead to unauthenticated and unauthorized users masquerading any privileged user, all the goals are at stake.

**Configuration.** Failing to properly configure server-side software, not keeping software up-to-date and not changing default passwords among others are
all security misconfiguration vulnerabilities that can be easily exploited to threaten all the security goals of this case study.

7.4 Potential Attacks

In this section we described some attacks related to the already mentioned vulnerabilities for this case study, and their concrete relation with it:

**Bypassing RBAC.** There are several ways to bypass the InfoBase RBAC, should the vulnerabilities be present: path traversal and URL manipulation to reach other repository (if their name is known) allow for bypassing access controls and directly reach confidential information. Another related attack is session fixation: in its simplest form, if an attacker can achieve to choose the session ID of a privileged user (by for example making the user to click on a URL with a chosen ID in the path), he can then bypass the access controls and impersonate the authentic user. This can potentially affect the Authentication, Confidentiality, Access control, Integrity and Accountability goals.

**Credential stealing.** Stealing of authentication tokens can be done via a man-in-the middle attack, should the related protocols be vulnerable. Also, tokens that are not session-dependent (i.e. do not expire) could be used indefinitely to forge a legitimate user’s identity. In particular, intercepting and modifying cookies could result in credential stealing attacks, resulting in a threat to the Authentication goal, and potentially to all other security goals should the credential stealing lead to an escalation of privileges (for example by stealing the credentials of a system administrator).

**Exploiting improper error handling.** As already mentioned in the vulnerabilities section, an attacker could exploit an improper error handling for example by forcing errors in the file-upload mechanism that escape into a privileged status. This is at the very least problematic for the integrity goal, since improper error handling could affect the stored data, but potentially threatens all other security goals if the escaped status is privileged.

**Cross-site scripting and injection attacks.** As for other use cases, there is a rich collection of cross-site scripting forgeries and injection attacks, like XSS, SQL, XML/JSON and XPATH injection. These attacks could be performed at any input channel of the InfoBase application and could result in confidentiality and integrity losses.
Attacks exploiting misconfiguration. These attacks include trying for the default administration password of any software used, exploit vulnerabilities of unnecessary open ports of a server and attacking known security problems of older software versions. It threatens all security goals potentially, depending on the seriousness of the misconfiguration.

7.5 Formalization

The Infobase ASLa++ specification, the code of which is given in Appendix D, can be summarized by the Message Sequence Chart (from now on, MSC) in Figure 28.

Before going into details of the formalization, we give a quick overview of the model, describing entities and the message exchange following the MSC of the model.

The model considers five different entities:

- Environment: the entity that encapsulates all the other entities;
- Session: for performing multiple executions of the scenario (for now there is only one session);
- User: the initiator;
- LoginService: this entity takes care of checking user’s credential for granting access to the repository;
- Frontend: this is the abstraction of the web interface that every user uses for performing every operation;
- Repository: this is the abstraction of the real storage entity.

The first communication step is between the User and the Login Service entities in which the User sends all the credential data (in particular, username and password), using an encrypted channel, to the Login Service that checks which kind of permissions the User must have.

After this first step, all further communications will be over a secure channel (with SSL) because after the User logged in, the Login Service entity knows the user. Then, after the first check, the Login Service sends the role together with username and password to the Frontend. With all this information, the Frontend can generate a cookie (using createCookie(RoleF, UserNameF)) and sends it to the User. Using this cookie, the User performs a new request (RequestU:=fresh();) and sends it to the Frontend entity together with his own cookie. The Frontend tests if the credentials are correct compared to
the request done by the User. Now, the Frontend entity forwards the request to the Repository which, after executing the request, will reply with a positive answer if the operation succeeded, and negative otherwise. Finally, the answer of the Repository is conveyed to the user by the Login entity.

After this introduction of the model, we will go into details of each entity and of each section of the Infobase formalization.

**User Entity**  
Like every entity, it starts with the keyword `entity` followed by the name (`User`) and two agents (`Actor` and `LoginService`) as parameters into the brackets. The `Actor` keyword is for referring to the considered entity (`User` here) and `LoginService` is the instantiation of the Login service entity used in the body.

Then, there is the `symbols` section that contains the instantiation of all the symbols used into the body. The names of these symbols are evocative, e.g., `CookieU: message` is used for the cookie and `Frontend: agent` for the Frontend entity.

```plaintext
entity User(Actor, LoginService: agent, UserNameU, PasswordU: text) {

symbols

CookieU: message;
RequestU, AnswerU: message;
Frontend: agent;

body {
% user sends user and pass to the LoginService
Actor ->* LoginService: UserNameU.secret_Password:(PasswordU);

% the Frontend entity responds to the login request (after LS
% forwards user and pass) with a cookie
?Frontend *-*-> Actor: ?CookieU;
```

Listing 7: Symbols section, User entity

Then we have the body section, which contains the message exchange of the User entity (Listing 8). The first part is for the Login phase, in which the User authenticates himself to the LoginService. At the beginning, the `Actor` (User entity) sends credential data to the `LoginService` entity (for the inline goal `secret_Password` please refer to Section 7.5) that replies with a new cookie `CookieU` whose value is caught by the `?` ([5]).

```plaintext
body {
% user sends user and pass to the LoginService
Actor ->* LoginService: UserNameU.secret_Password:(PasswordU);

% the Frontend entity responds to the login request (after LS
% forwards user and pass) with a cookie
?Frontend *-*-> Actor: ?CookieU;
```

Listing 8: Body section 1/2, User entity

The last part of the body concerns the request made by the User to the Frontend entity (Listing 9). Then there is `RequestU:=fresh()` for the creation of a new request. It is modeled using a fresh value since at this level of abstraction we do not need to know which is the real request that the user wants the Repository to carry out.
After this message creation, the Actor sends it to the User, concatenated with the CookieU to the Frontend via a secure channel (*->*). Then the Frontend does some operation described into Section 7.5.

```plaintext
%the user creates a new request
RequestU := fresh();
%the user sends to the fronted the request and a cookie
Actor *->* Frontend: CookieU, RequestU;
%the Frontend sends back to the user the answer of the repository
Frontend *->* Actor: ?AnswerU;
...
```

Listing 9: Body section 2/2, User entity

**LoginService Entity** According to the first line of the Listing 10 this is the LoginService formalization. As always there are some symbols defined into the corresponding symbols section: UserNameL, PasswordL and RoleL are text symbols for the user name, password and the role of the user, respectively. The UserIPL is the variable representing the user IP address and the LoginDataBase refers to the database.

```plaintext
entity LoginService(Actor, Frontend, User: agent) {  
symbols  
UserNameL, PasswordL: text;  
RoleL: text;  
UserIPL: agent;  
LoginDataBase: (text, text, text) set;
```

Listing 10: Symbols section, LoginService entity

In the body (Listing 11) of the LoginService entity, there is a select on the database so to extract the RoleL of the user. Then the LoginService will send all the information to the Frontend entity, which will perform the cookie based also on the information understood from the RoleL of the user.

```plaintext
body {  
%receives authentication data from the user  
select {  
%no matter on who is the sender of the data  
%this means that an intruder can easily  
%perform a brute force attack  
on(?UserIPL *->* Actor: ?UserNameL, ?PasswordL &  
%checks if the data are available in the database  
%select are "faster" than if
```

FP7-ICT-2009-5, ICT-2009.1.4
Project No. 257876
with the query below, we extract the role of the user
that sends us the credential data
\[ \text{LoginDatabase \text{--\text{contains}}(\text{\text{UserID}}, \text{\text{Password}}, \text{\text{Role}})) : \{ } \]
\[ \% \text{LS sends the role and the username to the Frontend entity} \]
\[ \text{Actor \text{--\text{--}} Frontend: } \text{Role}, \text{UserName}, \text{UserIP}; \]
\[ \} \]
\[ \} \] % end body LS entity
\[ \} \] % end Login Service entity

Listing 11: Body section, LoginService entity

**Frontend Entity** The entity Frontend symbol section (Listing 12) contains some instantiation, like in all the other entities. Two of these instantiations need to be clarified: checkPermission and credentialCookie. The first one is a fact and is used to check if the user that performed the request has the required rights, whereas the second one is used for cookie creations.

Consider now the body section (Listing 13). The first message contains the Role, the Username and the IP address of the user that the LoginService uses to create the cookie with the createCookie fact. Then, the LoginService sends the cookie to the user and the user creates the request (Listing 9) that this Actor receives in Actor \text{--\text{--}} UserIPF: CookieF. In the select scope, where the Frontend checks if the user has the required rights using checkPermission, it forwards the request to the Repository and then it forwards the response of the Repository to the user.
%after receiving the cookie the users makes his first request

%check if the user can do this request
select {
  on(checkPermissions(CookieF,RequestF)):
  if the user has the right credential, then the Frontend can
  send the request to the repository that will send back the answer
  Actor *->* Repository1: RequestF;
  Repository1 *->* Actor: ?AnswerF;
  the answer of the repository is sent to the user
  Actor *->* UserIPF: AnswerF;
  }
  %otherwise the user is a cheater...
}
% body closure Frontend
}
% end of Login Service entity

Listing 13: Body section, Frontend entity

Repository Entity The first line of the entity Repository defines name and parameters. Then there is the symbols section (Listing 14), that describes all the symbols used in the body. The first two symbols have message type and are used in messages. The elaborateRequest is an abstraction of all the operations that the real repository has to carry out to perform the requests of the user.

entity Repository1(Actor, Frontend: agent) {

  symbols
  RequestR, AnswerR: message;
  elaborateRequest(message): message;

Listing 14: Symbols section, Repository entity

After this first section there is the body of the entity. The first thing that we can observe is that all the communications are sent over a secure channel (*->*). The body starts with the communication between the Frontend entity that forwards to the Actor (Repository entity) the request of the user. The request ?RequestR is caught using the ? by the repository and then the request is elaborated by the elaborateRequest(RequestR). Finally, the repository sends the AnswerR to the Frontend in which there is the response of the elaborate action.
Sessions and environment  In this model, only one session is considered, since multi-session are not needed for the goal that we are considering now. More goals and more sessions will probably be added soon. In fact, as we can see in the code below (Listing 16), there is the body of the Environment entity that shows one session with the constraint that \( !\text{User} = \text{LoginService} \). The latter assertion means that we are sure that the User and the LoginService entity must not have been the same value.

More interesting than the single session is how we instantiate each entity. The Listing 16 is showing that there are four different instantiations, each one has a \texttt{new} followed by the name of the entity and the values of each parameter that are the values of the entity involved in that particular entity.

For example, in \texttt{new Repository1(Repo1,Frontend)} there is the \texttt{new} keyword to start the instantiation followed by the name of the entity that we want to instantiate (\texttt{Repository1} here). Then, between brackets, the first value (\texttt{Repo1}) is the value of the \texttt{Actor} entity used into the \texttt{Repository} entity, followed by \texttt{Frontend} that is the instantiation value of the second parameter of the Repository entity.

Goals  Since this is not the final model and it is continuously changing with more features and details, we have only modeled one security goals up to now.
The body section of the User entity contains the inline constraint
\[ \text{Actor} \rightarrow \text{LoginService: UserNameU.secret\_Password: (PasswordU)}; \] (and the corresponding one into the User entity) that is used in the goals section (Listing 17) to ensure that the confidentiality of the password exchange between the LoginService entity and the User entity is guaranteed.

```plaintext
157 goals 158 % the Password must be a secret between the user and the LoginS 159 secret\_Password: (_) { Peter, LoginS};
```

Listing 17: Goals section
1. user, password
check permissions
2. role, user, password
3. cookie
create cookie
check db
4. cookie, request
5. request
6. answer
7. answer

Figure 28: Infobase MSC
8 eHealth

8.1 Application Scenario Description

8.1.1 Overview

There is a strong need for a flexible information infrastructure that facilitates innovation in wellness, health care, and public health. Health ICT has been called to:

- support the communication between the health care providers, third parties, and patients,
- make available to care personnel the patients’ medical history and present conditions (test exams, etc) in the moment they need it,
- support clinical decision making for the prevention and treatment of illnesses,
- be used to acquire population data for public health, and
- help improve the quality of public care and of research.

Both the lack of information about the medical history of a patient and the unavailability of medical guidelines or decision support has been the cause of improper diagnosis or treatment.

It is therefore not surprising that eHealth is an area of rapid innovation. Many different solutions have been advanced and will continue to emerge, both for information systems, which manage patient data or other medical information, and for computer-aided medical decision support systems. And although the proposals have different formats, models, and systems are disparate, quite a few of them will co-exist for at least the next couple of decades, and there is a strong pressure to integrate them in a coherent way.

This use case is based on mashup systems that on the one side create and use electronic Health records (private patient information, see below), and on the other hand aggregate other functionalities like decision support for the practitioner, analysis of images, billing systems, etc.

Management of basic Electronic Health Records. We use the term Electronic Health Record (EHR) in a very broad sense: it may include any information created by, or on behalf of, a health professional in the context of the care of a patient. More precisely, a record can consist of:

- Computerized
• Laboratory reports,
• X-ray films, other imaging records or photographs,
• Printouts from monitoring equipment,
• Correspondence between health professionals,
• Scans of handwritten medical notes, and
• Videos and other recordings, including audio recordings.

For the purposes of our scenario, we assume that EHRs are collected and maintained both by general practitioners in their office systems, and also at the different wards or departments in a hospital. The entity that collects and manages the EHRs (the system in the practice of the doctor or in the ward or department of the hospital) will be called the administration domain of this particular collection of EHRs.

Note that, for the purposes of this project and for the sake of advancing a concrete scenario to be built and tested, we do not consider the case where hospitals group together to build large circles of trust that share, under legal regulations and self-imposed rules, patient information. We do not either consider the case of Personally Controlled Health Record, for instance [11], Indivo X or Microsoft’s Health Vault. In that model, the personal vaults are built with the purpose of securely collecting all medical data relating to a patient. The idea is that the patient has, at least in theory, the direct control over the policy rules determining the access to his medical data.

Policies. We assume that there are rules, or access control policies associated with the single EHRs that describe what is allowed to be done with the information: who is able to read the data, to write it (and how), delete it (and when), and for which purposes an EHR is available to whom.

One approach useful to formalize the policies applicable to EHRs, is close to the so-called BMA model [1], where each record has an explicitly or implicitly associated Access Control (AC) list. For other related work, see [21] and [24].

The administration domain (that is, the hospital ward or the medical practice) is responsible for implementing and managing the policy enforcement mechanisms and placing the correct enforceable policies and meta policies (that is, the policies that regulate who is able to read or write the policies to access the EHRs) in the system. It is still possible in this setting that the administration domain chooses to give the patient full control over the policies on his EHRs (within the limits and restrictions that regulation
may require), by providing an interface to the patient to express his policies in detail, but we do not assume this to be necessarily the case. As discussed below, in any case, we assume that for several types of actions, it is required to have patient consent.

Information may pass between the different administration domains (wards, even in different hospitals) in a controlled manner, in the form of summaries (such as referrals or discharge letters to General Practitioners, which are also EHRs).

Health care regulations, like HIPAA, are complex, difficult to understand and to implement correctly; enterprises have difficulties in verifying and demonstrating compliance of their systems.

Electronic health records should support clinical research [18, 24, 19]. But, as discussed in [20], HIPAA complicates the research process. This may apply too to other privacy regulations. To make patient data available to research support services, overhead in terms of resources and time are necessary. This is due, among other reasons, to concerns about the security of patient medical information, which makes health organizations more hesitant to let researchers have access to the patients’ information, especially via electronic transfer. To alleviate this, a strong assurance is necessary that the patient has provided consent and the data has been anonymized or has been aggregated correctly and has been transmitted securely to the intended recipient(s).

8.1.2 Scenes Description

As a point of departure, let us consider the following situation: Dr. Davis, a family doctor with his own practice or a doctor in a hospital, is treating patient Alice, and, in that role, he has access to a local EHR repository (in the medical practice or in the hospital) containing the patient data of Alice. Dr. Ellis and Dr. Finch work in the same hospital or share a medical practice with Dr. Davis and, in principle, have access to the same EHR system but are not authorized to read the EHRs of Alice. Now, some typical single scenes are:

- **Mashup: reading third party data.** Dr. Davis wants to compare online the data of Alice with reference data, for the purposes of differential diagnosis. Further, he wants decision support on recommended tests or the best therapies or pharmaceutical drugs and advice on typical progress of the condition or possible complications. He may also want to add data sources from diverse sites such as pharmacies or the insurance company to obtain
the best possible affordable treatment or medication. For this purpose, Dr. Davis will use an application that provides a visual area (which we will call a “window”) containing an integrated user interface, and inside this area he will find data of possibly many types, text, figures, images, recordings, etc. The data will have different origins: it may have been provided by a local or remote intelligent support system, by the insurance company or another third party, or the data may have been extracted form the local EHR database. Dr. Davis will seamlessly move inside this one window from a region where the data of Alice is visible, to regions where data from the other sources is available. Moreover, by simply “clicking” or “marking” some text in one of the regions, the information in other regions may change accordingly. For instance, when he marks the symptoms or the test results of Alice, the support system will pop up possible suggestions for diagnosis or medication on the screen, and the insurance company, the pharmacies and other parties will provide information about affordable medication or treatment.

A similar situation, which we want to subsume into the same scene, occurs when Bob, a patient with a chronic illness wants to obtain from home information about his situation, in the hope of accelerating his treatment and cure. He is able to authenticate into the system, using his own Identity Provider, and use a similar visual interface to view information relevant to his case and his particular situation.

- **Mashup: complex interaction with third parties.**
  Dr. Davis plans to order a particular treatment (a physiotherapy, or a special therapy at a tertiary care medical service center) or a special medicine, which could be covered or not by the insurance plan of the

---

18For our purposes it is almost irrelevant which is the precise way the information is presented in the window. Logically, the visible elements can be distinguished by type and origin. For simplicity, and also to avoid unnecessary complications that are not in the scope of our discussion, we assume that the whole screen contains only one window which is divided into regions where data of the same type and same origin are presented. But it is important to keep in mind that instead of placing information in different regions, modern visualization technologies can be used to present the information, for instance overlaying the information of different sources into the same area. Thus, the user of the system may not clearly know where the data that he is viewing comes from, and this may also lead to confusions: the doctor may interpret some data from an external source as being the patient records in his database. This type of presentation problems must be addressed with much care, but will not concern us any further in the project.

19It is also possible to overlap the patient data of Alice within reference diagrams, figures or images, but as discussed in a preceding footnote, extreme care must be taken, that the doctor does not confuse patient data of Alice with other data.
patient. The same system can be used for this purpose, automating the negotiation with the insurance to cover the costs and authorizing the planned treatment.

- **Patient Consent.**
  Dr. Davis wants to submit patient data of Alice, properly anonymized, to the health database application. This information will be used there for purposes of medical research, promoting and advancing public health care. For this external transfer of EHRs, Alice’s consent is required. Dr. Davis asks the system to obtain patient consent, using the same interface. The system contacts Alice asking for the consent. Now Alice must first authenticate to the system, using her own Identity Provider, which we assume to be trusted by the system. Now Alice can provide her consent to the system and Dr. Davis can continue his work as he planned. The same basic situation arises when Dr. Davis wants to hear the opinion of Dr. Ellis, who initially is not authorized to read the EHR of Alice. To get this authorization, the policies require that Alice provides consent.

- **Emergency Case.**
  Dr. Finch sees Alice fainting in the waiting room. Dr. Davis is not immediately available, so Dr. Finch decides to access the EHR. In general, all medical practitioners that have access to the system may read Alice’s EHRs in case of emergency. In our particular case, Dr. Finch has a special access to the EHRs of Alice, by declaring to the system that now an emergency situation is happening, and a special logging mechanism is then used to record this fact and all actions that Dr. Finch does in this context. The rationale is that doctors are generally thought as trustworthy, and the sacrifice of any patient-specific access restrictions is compensated by extra accountability (in the legal sense).

- **Policy Administration.**
  Authorized privileged users are able to create, view and sometimes modify the parameters (like groups, roles, or assignments in the case of RBAC-like policies) of the privacy policies established by patients, care delivery organizations and jurisdictions. Access to those parameters is regulated by the meta policies.

- **External access to central system and repository.**
  In the case of a server-side mashup, third party applications may be given the rights to connect to the EHR central data server and even to
access parts of the repository across a standard application programming interface. This is useful to allow the external application (for instance, a financial application, an insurance or a pharmacy) access directly some patient data that they are allowed to know and need for their own purposes. Of course, a challenge here is that the external applications are not able to access more data than what they are allowed to. This is particularly difficult to control in the mash-up scenario because currently there are no policy enforcement points in the browser that are able to effectively control what data is accessed by which applications (as opposed to a traditional web-service environment, where the web application service could for example check OAuth tokens or other authentication credentials to cope up with this problem).

- **External access to user browser and content.**

  In the case of a client-side mashup, third party applications may run JavaScript or other programs or active elements on the browser of the user (medical practitioner or patient) who is connected to his trusted EHR central data server. In this way, a third party may access data on the browser of the user, perhaps belonging to a different origin, or to patient data in the browser. As in the previous scene, the purpose is that external applications access patient data that they require. (The threat, again, is that the external application uses JavaScript to access data that she is not allowed to access).

### 8.1.3 Architecture Description

At the moment of writing this deliverable, the final architecture of this prototype has not been decided, nor the implementation details. We will therefore not explicitly list the technologies used. There are however two main architectural choices (that could eventually also coexist), namely server-side and client-side mashups.

The basic architecture of the system, using a server-side mashup, is shown in Figure 29. The central component is the eHealth Server. It composes the functionality of different back-end applications and provides the composed functionality to the end users, which can be medical practitioners or patients (or sometimes other stakeholders).

Another possible architecture to implement the system, using a client-side mashup, is shown in Figure 30. Now it is the browser the one that composes the functionality of the various back-end application servers and presents the composed functionality to the end users.

It is also possible to use a mixed architecture: a client-side mashup can
also access an application coming from a trusted EHR Server which is itself combining information and functionality from different applications. This makes sense for example if the EHR Server performs calculations with data coming from different sources that would be too expensive to run on the client-side, whereas lighter applications could be allowed to run as JavaScript on the client browser.

8.2 Security Goals

Since storage and communication of data in an EHR comprises sensitive personally identifiable information (PII) within patient data, each of those functions needs specific security and access management requirements to be considered and implemented.

The eHealth Record scenario is characterized thus by several security requirements, like Privacy, Confidentiality, and Authenticity of Electronic Health Records (EHRs). In order to be considered compliant to these re-
Figure 30: Basic architecture of an eHealth Client mashup.

requirements, the abstract model must meet a set of security properties (that will be subsequently tested also in the real system by means of test cases).

Although there are no complete models of the EHR case studies at this time, nevertheless it is possible to define a list of relevant goals that any model should meet.

As already mentioned, at a high level, the security requirements for health records and systems are: the confidentiality of patient data must be enforced, the authenticity and integrity of EHRs must be secured, the system and processes must be reliable and available. Moreover, special scenarios, like the emergency case, the user consent to forward information to other parties, and the anonymization and controlled declassification for research purposes must be supported. In more detail, our goals are:

- **Confidentiality**: In general, both sensitive medical data transmitted over the network and data stored should be confidential. In particular, patient data should be confidential against third parties unless specific patient-consent is given. This is also the case when, for a second opinion, a further medical practitioner obtains access to the EHRs of the patient (except in emergency or in the case of statutory exemptions). More specifically, Patient records must be protected from eavesdrop-
ping during communication and this usually implies that communication must be encrypted. Additionally, EHRs should be protected at rest, when stored on servers, on devices or on clients. This goal is particularly important in the case of eHealth mash-ups: since different applications work on the same data space, it is important that the separation techniques in place do enforce, as expected, that the applications cannot access into the data of other applications, outside of the defined interfaces.

- **Privacy:**

  In addition to the more general goal of confidentiality, we plan to consider the following specific and stronger privacy goals. Although it is not clear whether we will be able to fully model all of them in ASLan++, we want to have a diverse collection of goals as a starting point for future formalization efforts.

  1. **User Choice and Consent:** In our case, the patient has the choice that some of his medical personal information is being sent to another party for a particular purpose, like a second opinion or for research purposes. This is called *Patient Consent*. This goal can be expressed with an LTL goal, having implemented in the model suitable access control policies via Horn clauses and dynamic facts.

  2. **Notification:** the patient has to be informed about access events to his Electronic Health Records (EHRs). This is a functional goal that is likely to be abstracted in ASLAN++.

  3. **Data Retention:** a doctor that provides a second opinion in a particular case should, of course, be able to read the pertinent EHRs, but should not keep a copy of them after giving his opinion. This goals is more of a ‘usage-control’ nature, and therefore more challenging to be modelled.

  4. **Extra Protection of Highly-Sensitive Data:** Some particularly sensitive information, like socio-psychological EHRs, must be specially protected and can only be read by a small number of medical practitioners that indeed need to know this information. Also the information relating to VIPs (celebrities) requires special protection rules. Even the fact that a particular patient is being treated in a certain hospital can be treated a secret in many circumstances. As said in the introduction, this could be considered a confidentiality goal, but we wanted to highlight it separately in this section together with other strong and specific privacy goals.
Other privacy properties like *Purpose of Use* (the user is informed about the intended usage of his data, and the data is used only for such purposes) and *Data Quality* (the user is able to correct any incorrect information in his data), as well as those related to Procedures, Training, Awareness and Standard of Conduct, Incident Management, Auditing and Enforcement are out of the scope of our investigation (mainly because they lie on a business-process level of abstraction and not in the software level, that is the main scope of this project).

- **Authenticity of EHRs:** It must be verified that the EHRs have been created by the claimed Person at the claimed time (this information is included in the EHR itself) and using the procedures, methods, workflows and systems approved for such purposes.

- **Integrity of EHRs:** Data stored in the EHR must remain complete and unaltered unless update operations are performed by authorized users. Their logic (workflows and processes) should also be maintained.

- **Dynamic Access Control Policies (Authorization):** In principle, EHR can be only accessed by authorized users. However, there are exceptions (i.e, emergency case) in which a medical practitioner access the EHRs with relaxed access control policies with special provisions for logging the activities.

- **Accountability:** In the case of eHealth, every access to an EHR must be logged. This is particularly important when a medical practitioner accesses an EHR with relaxed access control policies in the case of emergency. The special provisions for logging activities take place can be expressed through an LTL goal. It may be sufficient to check that if an emergency occurs (implying relaxed access policies to a record), then extra logging activities must be performed (this may be represented by a special fact), and this can be modeled with an LTL goal. In addition, each time an access occurs, a log is also immediately done.

### 8.3 Potential Vulnerabilities

The vulnerabilities of this case study can be classified in mainly two categories: those concerning classic web applications and services and the ones arising in the context of mashups. Nevertheless, most of the known vulnerabilities in mash-ups could be roughly classified into the first category. We point out in the following which of these are particularly relevant in this context.
Authorization. Authorization flaws lead potentially to privacy and confidentiality leaks. For example, a component of a mash-up could be authorized to read statistical data from the EHR, but not single data records. Could a malicious application exploit an authorization flaw vulnerability, it would potentially escalate its reading privileges. Another example would be a user with the authorization level of a nurse or researcher that could maliciously obtain the authorization to perform activities only allowed to physicians. In this case he could not only gain access to confidential data, but also execute integrity-undermining operations on the EHR.

Authentication. Similarly to the previous vulnerability, an attacker could benefit from broken authentication by obtaining re-usable identification tokens and impersonate other users with higher privileges. A critical point where such a vulnerability plays a role is in patient-consent. A flaw on this mechanism could allow an attacker to forge future consents.

Concurrency. Concurrency flaws may be abused potentially not only to gain privileges on the system by exploiting some buggy execution under race conditions, it could also lead to DoS attacks. This sort of attacks are very likely to be exploited by malicious components in a mash-up, should their dynamic data not be properly validated (see below).

Input validation. Traditionally, care must be taken whenever validating input data to web applications (via forms, URLs etc.). This vulnerabilities must specially taken care of in the eHealth case study context. Moreover, since components of a mashup interact with each other and potentially their content changes on demand, new vulnerabilities arise. The lack of proper validation of this dynamically generated content goes beyond the traditional data validation.

8.4 Potential Attacks
Most of the known attacks to web-applications are relevant for this scenario, independent of the architecture chosen (server-side vs. client-side mashup). In this section we give example of relevant attacks, that would follow by exploiting the already mentioned vulnerabilities. We will keep a high level of abstraction in the description of this attacks, since the concrete technologies for the mash-up have not been so far established (whether services will communicate using XML, the GUIs will be browser using Javascript or Ajax etc.).
Path traversal/URL manipulation. Given that users have access to a structured file organization where paths are used (this is also the case if the users access the EHR via a web-browser), a flaw in the authorization mechanisms could be exploited by changing the paths and accessing configuration information or information of other users. This threatens mainly the Privacy goal, but could also lead to a serious compromise of Integrity, should the attacker escalate its privileges as a consequence of this attack.

Session fixation. If an attacker can fix the session ID (via a mashup component, e-mail with malicious link or however), he could gain the privileges of the victim in the eHealth system. This could happen inadvertently if the data coming from mashup components is not properly validated (see below). Should the compromised victim be a regular user, Privacy of his data is at the very least compromised, but the attack could affect mostly all security goals if the attacked user has administration privileges.

Interception of identification tokens. If identification tokens, such as those used in the patient-consent protocol or log-in to the system/service are reusable, an attacker could easily forge an identity in the system and gain unauthorized privileges, compromising the Privacy and Authentication goals.

Cross-site scripting and Injection. Besides traditional cross-site scripting and injection, in the mashup context the input data can spread from one component to another within the mashup. Even if the components are supposed to share some data (for example patient records, or summaries of patient data for statistical purposes), a lack of proper validation could mean that also some logic (in form of Javascript, Ajax, SQL etc) can be maliciously forced to be executed by the browser or eventually by a particular back-end service. As a consequence, private data may leak, but also data could be modified, resulting in an integrity loss.

DoS attacks. DoS attacks can arise by exploiting race conditions on multiple connections, but also on the concurrent nature of the mashup. Indeed, the parallelism of the various components is a new dimension of complexity (with respect to traditional concurrency in web applications). On the other hand, not properly validated malicious code could abuse system resources to cause a client-side or server-side DoS. This would affect the Integrity of the workflows.
9 Evaluation Criteria for SPaCIoS Tools

Evaluation criteria for testing methods/tools are based on the data that can be measured in experiments. In this section, we list a number of evaluation criteria that can potentially be used for assessing the quality of the SPaCIoS testing methods/tools. The list is not exhaustive, and perhaps some of the listed criteria are not readily applicable to SPaCIoS. We will decide on suitable evaluation criteria for SPaCIoS in later stages of the project.

In SPaCIoS, two kinds of security testing methods/tools are investigated: property-driven security testing (WP 3.2), and vulnerability-driven security testing (WP3.4). In particular, testing the functionality and performance of the SUV, although equally important, is not addressed in SPaCIoS. Therefore, we refrain from listing evaluation criteria that concern such tests.

We divide the list of evaluation criteria into “absolute” metrics and “comparative” metrics. Absolute metrics can be applied to a stand-alone testing method/tool, while comparative ones only allow us to compare a testing method/tool to another one.

9.1 Terminology

Before listing the metrics to evaluate SPaCIoS tools, we define the meaning of some terms used in this section to avoid confusion afterward. In the following, we adopt the standard testing terminology with regards to faults and failures.

Faults and failures. In testing, a failure is any deviation of the software system from its expected behavior as observed by a test case. Therefore, a successful attack exhibited on a software system represents a failure in the system. This is because the attack allows us to observe that the system deviates from its expected behavior, namely (at least) one of its security requirements is violated. Failures are caused by faults, i.e. vulnerabilities in the software. A fault may or may not lead to a failure: in terms of security testing, a vulnerability may or may not be exploitable. Testing concerns the observable behavior of software. Therefore, detecting failures is the main activity of testing. Strictly speaking, software faults cannot be found via testing: the existence of faults in the software is implied if failures are exhibited during testing. Thus, a fault in the system is “detected” by observing the resulting system failure.

Attack traces and test cases. An attack trace is a sequence of messages together with, for each single message, its sender and receiver, leading
to a security goal violation. In order to transform an attack trace into a test case, the communicating agents must be separated into two parts: components that are simulated by the test driver and system agents that become part of the system under test (SUT). Messages sent by the tester are input messages (i.e., the test execution engine must provide them as an input to the SUT). The other messages are output messages emitted by the SUT. An attack succeeds if it can exploit one or more vulnerabilities in the SUT to violate a security goal.

**Test verdict.** A test case is a finite behavior over system stimuli (inputs) and system responses (outputs). Each path through this behavior leads to a test verdict of either pass or fail. The verdict is pass if no failure could be observed and fail otherwise. In that sense, if an attack succeeds, the SUT fails the corresponding test case. The expected behavior of a test is the behavior that leads to a pass verdict. A system is deemed correct if it passes all tests implying that it fulfills given security goals and no vulnerabilities could be detected. Otherwise the system is erroneous.

### 9.2 Absolute metrics

1. **Fault detection power:** In a controlled experiment, how many known vulnerabilities (“real” or seeded security bugs) in a given SUT are detected by the set of test cases.

2. **False positive rate:** Number of test cases that produce a fail verdict, but the SUT does not violate any security goal. This can happen because of differences between the model used for testing and the SUT or limitations of the test derivation algorithm used. Note that property-driven testing in SPaCIOs hinges upon a model for the SUT. The fidelity of the model inevitably affects the precision and quality of the tests.

3. **False negative rate:** Number of test cases that produce a pass verdict, but they correspond to vulnerabilities that are known to exist in the SUT (e.g., seeded vulnerabilities). As for false positive rate, the reasons for this can be lacking fidelity of the model, weaknesses of the mechanism used for bridging the abstraction gap between abstract test cases and executable ones, or bugs in the testing toolset.

---

20The common use of “a test fails” is ambiguous in the security context as it could mean either the attack does not succeed (from the attacker’s point of view) or the SUT is vulnerable (from the developer’s point of view). Here, we chose the latter meaning and we use the non-ambiguous expression: “A SUT fails a test.”
4. **Number of generated tests:** What is the size of the generated test suite in terms of test events (interactions with the SUT)? We remark that this criteria depends on the quality of the model that is input to the SPaClOoS toolset.

5. **Test generation and execution time:** What is the time needed to generate a test suite from a model and what is the time to automatically execute all generated tests?

6. **Efforts spent on various testing phases (in person hours):** For a given SUT: modeling, test generation, test execution setup, test execution, test result analysis. All those phases are mandatory to actually test a SUT by using SPaClOoS tools. Quantifying the time needed for each of them will allow us to focus on improving the longest phases and show evidence of the tool usability. This criteria is sensitive to the experience and knowledge of the person who performs the modeling and uses the tool. In other words, the results of this kind of experiments may be very hard to generalize and potentially heavily biased.

7. **Average effort to locate faults:** The interest behind using a test tool is not just to find a failure but to fix the origin of the detected failure. The average time needed to locate the fault at the origin of the failure depends on the information given by the testing tool. Thus, this metric is important to evaluate if reports from the tools are precise enough or if it is necessary to give more feedback to the user. This criteria is sensitive to the fidelity of the model used for testing and also on the experience and knowledge of the person who uses the tool.

A few notes are due:

Firstly, metrics 1, 2 and 3 require an understanding of the vulnerabilities (system faults in classical testing) that we want to address in SPaClOoS and that the tool shall be able to detect. A list of typical vulnerabilities addressed in SPaClOoS is available in [22].

Secondly, usability of SPaClOoS tools can be assessed using a questionnaire for the users of the tools. Note that in particular for vulnerability-driven security testing, the purpose is to facilitate the activities of white-hat hackers.

Thirdly, instead of counting the number of generated test cases, it is sometimes more interesting to measure the number of different test cases according to which attack/vulnerability they target. As two test cases could be two instances of the same attack or exploit the same vulnerability, it is important to assess the variety of a test suite rather than its size.
9.3 Comparative criteria

It would be desirable and interesting to use absolute metrics for comparing the quality (metrics 1–3) and the usability (metrics 4–6) of the SPaClOoS tools to other tools:

- With respect to property-driven security testing, we note that for a fair comparison, however, the input (model, property, vulnerability description) given to SPaClOoS and the tools it is compared with should be identical (or precisely comparable). This is a challenge: different tools often accept inputs in different languages, and the input to the analysis tool has a fundamental impact on the quality of the results. Indeed, very often knowledge of the inner workings of an analysis/generation tool helps build models that the tool can cope with. Therefore, there is a risk of bias when we use optimized models for SPaClOoS and non-optimized models for the other tools. To avoid this problem, we can compare SPaClOoS with random testing: we can use the same input for both SPaClOoS and for random test generation. Below, we come back to this point.

- With respect to vulnerability-driven security testing, it seems desirable to compare our tools with existing methodologies, e.g., the OWASP Security Testing Guidelines. For this kind of tools, comparing with random test case generation may not be a reasonable choice, because the current penetration testers are performing penetration testing based on their knowledge and experiences rather than doing it randomly. Unfortunately, this approach also exhibits the subtle flaw that the OWASP methodology can be followed in various shades of quality. However, comparison of the SPaClOoS tool with other commercial or freely available test tools and methodologies is important as it shows the improvements of the SPaClOoS project over the current state-of-practice in security testing.

On comparison with random testing. It is a standard procedure to compare the fault detection power of tests generated by tools being evaluated to that of randomly generated tests. This means that, in addition to generating tests from properties/attacker models, we will have to generate tests randomly. There is a plethora of ways of randomly generating tests: e.g. uniform random input, random input based on usage profiles, uniform random walks through the model, random walks through the model that make different paths be chosen uniformly, etc.
Then, as a criterion for the comparison, we take the fault detection power: we compare the fault detection power of the SPaCIoS tests to that of the random tests. We remark that the comparison criterion must be sufficiently different from the test selection criterion used to generate tests. Otherwise, we would not be surprised by having high rates according to that assessment criterion. Thus, we deliberately chose to not consider mutants of the SUT (i.e., using mutation operators to automatically introduce vulnerabilities into the SUT) for the tool evaluation as we already use mutation operators for the test generation and the risk of an overlap between the mutation operators is too high.

As a final note, a pitfall of using random testing for security is that random testing can be poor at detecting security problems. There might be a situation in which we always detect zero problems with random testing which would question random testing as a basis for assessing the quality of SPaCIoS tools.
10 Preliminary Assessment

This section reports preliminary testing activities, performed to assess the technologies and tools developed in the SPaCIoS project so far. Applications scenarios WebGoat and SAML SSO are involved.

10.1 Assessment of mutation-based testing within WebGoat

In this subsection we describe a security mutant operator to transform the secure model of a WebGoat lesson into a security mutant. The mutation operator is defined in such a way that the resulting security mutant corresponds to the presence of a specific vulnerability in the WebGoat lesson (here, an authorization flaw that allow to bypass the data layer). For our described mutant the model checker will report an abstract attack trace that we instantiate to execute it on the SUT. We describe how we bridge the abstraction level and how we verify that the execution of the test case was successful.

10.1.1 Security Mutant

The model described in Section 2.5 can be used as input for a model checker. In our example we use the Cl-Atse model checker [23] to verify whether the specified security properties of the model are satisfied. Three goals are described in the model: two in the Session entity (see Listing 3) and one in the Environment entity (see Listing 5). The last security goal is the one we are interested in here. It states that the intruder i can access to a profile only if he is allowed to view it.

Our ASLan++ model is secure, meaning the CL-AtSe model checker does not find any attack in the model. In order to make use of the model for testing purposes we provide a mutation operator that slightly modifies the secure model into a potential insecure mutant. In our example the mutation operator removes the canView check at the server agent in line 47 of Listing 4. After applying that mutation operator the modified part of the model looks like as follows:

```plaintext
12  % ViewProfile action
14  \& cookies(Actor)->contains((?U, ?Cookie))
15  XXXXX \& ?U->canView(?A)
16  ): { 
17      select{ on(A->hasProfile(?Profile));{} } % get the profile of A
18      Actor *-* U : A.Profile;
19  }
```
Thereafter the mutated model is checked by the model checker. Giving this model to the Cl-AtSe model checker, an attack is found and reported, as listed in Listing 19.

<table>
<thead>
<tr>
<th>VIOLATED:</th>
<th>secret_profiles[A=moe, P=moeProfile] % on line 114</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESSAGES:</td>
<td></td>
</tr>
<tr>
<td>1&lt;curly&gt; -&gt;* webServer : login(curly, password(curly, webServer))</td>
<td></td>
</tr>
<tr>
<td>2webServer &lt;-&gt;* &lt;curly&gt; : n11(Cookie)</td>
<td></td>
</tr>
<tr>
<td>3&lt;curly&gt; -&gt;* webServer : n11(Cookie).viewProfileOf(moe)</td>
<td></td>
</tr>
<tr>
<td>4webServer &lt;-&gt;* &lt;curly&gt; : moe.moeProfile</td>
<td></td>
</tr>
</tbody>
</table>

Listing 19: Abstract Attack Trace

Section **VIOLATED** reports the security policy that is violated. In our specific case the model checker reports a violation of the `secret_profiles` policy for `moeProfile`. In addition, the **MESSAGES** section lists a message sequence that drives the model into a state where the reported security policy is violated.

Here, four messages are needed to violate the policy. In the first step, a user **curly** — the compromised user curly — logs into the server by sending the `login` message, which contains the shared password. The server agent (**webServer**) acknowledges the request by sending back the shared cookie. As a third step, the user **curly** tries to request the specific profile `moeProfile` that he should not be allowed to view it according to the security policy. The model specifies that `moeProfile` is only viewable by its user **moe**. Due to the absence of the critical check at the server side, whether the requested profile may be viewed, the request is accepted and the profile `moeProfile` is sent back. The fact that the user **curly** gets `moeProfile` is considered as an attack because of the specified security policy in Listing 5.

### 10.1.2 Instantiation of the Abstract Attack Trace

What we have achieved so far is defining a security mutant operator that describes a possible vulnerability at the implementation level that leads to the violation of a security goal. The security mutant operator is defined at an abstract level. Once a model checker reports an abstract attack trace, we need to verify as a next step that the described vulnerability is indeed present in the considered SUT. Therefore we take the abstract attack trace from Listing 19 and instantiate it so that we can execute it in our SUT.

For the instantiation we simulate the compromised agent **curly** and run **webServer** as the SUT. That means that the abstract attack trace is instanti-
ated in such a way that our test framework generates messages from curly to webServer and checks messages from webServer to curly if they correspond to the expected responses. If we consider a deterministic behavior from the Web server, then receiving a non-expected message from webServer means that the attack described by the test case is not reproducible. In case of nondeterminism, further eventualities must be considered (see, Section 10.1.3 for some examples).

The instantiation process is split in two steps. The level below the abstract modeling language defines a set of still abstract, but common methods for dealing with Web applications. These high-level web service methods are designed from the viewpoint of an end user and the mapping is application dependent. A non-complete list of such functionality are:

1. Accessing pages by typing their URLs;
2. Entering text in text fields;
3. Following links;
4. Triggering a click event on buttons;
5. Selecting a value from a dropdown list.

In our example, the first message (line 5 of Listing 19) corresponds to the login action of curly at the webServer. This action is mapped to the following sequence of abstract web service methods where \( \circ \) denotes the composition of functions:

\[
\text{login}(\text{curly}, \text{password}(\text{curly, webServer})) = \\
\quad \text{clickButton}(\text{login}) \\
\quad \circ \; \text{type}(\text{password}(\text{curly, webServer})) \\
\quad \circ \; \text{selectDropdownItem}(\text{curly})
\]

In this mapping the testing expert provides information, e.g., the caption of the element in the dropdown list that has to be selected, the name of the text field and the text that has to be typed, or the caption of the button to be clicked. The second message (line 6 of Listing 19) is a reply from webServer and has therefore to be checked. To do so, we map the message to the abstract web service method checkCookie. This check can be performed using different granularity levels. It is the task of the testing expert to define when a cookie is considered as valid. The third message deals with sending the viewProfileOf message. This abstract message is mapped to the abstract web service method sendRequest. So far the test expert has to provide the
payload of this request that consists of the actual function name and the
profile ID of user moe. As a final step, the response of webServer has to be
checked if it contains the profile moeProfile. This message is mapped to
the abstract method checkResponse where the test expert provides a unique
criterion to identify the profile.

The second (and final) step in the instantiation process consists in trans-
lating the still abstract messages to real API calls. In our example, we
make use of two different frameworks, both operating at a different level
of abstraction. One of these frameworks is the Selenium Web application
testing system. Its API addresses actions at the HTML layer (triggering
click events, typing text, accessing URLs, ...) and therefore binds the ab-
tract Web service methods to HTML actions. Since it may be the case that
some parts of the attack trace are not executable on the basis of HTML
functionality, the Apache HTTPComponents project is used in addition to
Selenium. An example for the need for an HTTP component is the follow-
ing one: Let us assume that parameters of a method call are determined
by selecting elements in an HTML list. In this scenario only parameters
whose corresponding elements are available in the HTML list can be used
for method calls. The specified attack trace in Listing 19 requires that a
parameter is used for the method call that is not available in the HTML list.
Such a method call cannot be triggered at the HTML level by using the API
of Selenium. In order to execute a method call with a parameter that has no
representative at the HTML level, the test engine makes use of the HTTP
component library. This library enables the test engine to directly generate
HTTP requests that would not be possible at the higher HTML level.

To give an example how abstract web service methods are mapped to API
calls in the underlying testing framework, we consider two mappings. In the
mapping from messages in the abstract attack trace to abstract web service
methods, we made use of the method clickButton. Since a button is present
as an HTML element, this action is mapped to the following Selenium API
call:

```java
1 driver.findElement(By
2 .xpath("//input[@type='submit' and @value='Caption']")
3 .click();
```

Listing 20: Source Code Mapping

In this piece of code, driver corresponds to the Selenium connection
object that is responsible for sending requests to WebGoat and receiving

---

21 http://seleniumhq.org/
22 http://hc.apache.org/
responses. Obviously such a driver component has to be configured and instantiated properly before it can be used as listed above. Furthermore, Caption denotes the label of the button.

The second example shows how the abstract web service method `sendRequest` is mapped to executable source code. `sendRequest` is mapped to the HTTP component because we cannot make use of any HTML element. Therefore the corresponding source code looks as follows:

```java
1. httppost.setHeader("Content-Type",
                          "application/x-www.form-urlencoded");
2. httppost.setEntity(reqEntity);
3. HttpResponse response = httpclient.execute( httppost );
```

Listing 21: Source Code Mapping

Identical to the driver component, the `httppost` components needs to be correctly initialized before it can be used. The HTTP header is added by the `setHeader` method and the actual payload is stored using `setEntity`. Then the request is sent and the corresponding response is received from the WebGoat application.

According to the described abstraction layers, the instantiation of an abstract attack trace, in order to be executed, happens in several steps. Terms and expressions of abstract attack traces are mapped to a message or a sequence of messages of the abstract web service methods layer; abstract web service level methods are mapped to API calls of the underlying frameworks. The splitting of the instantiation into several steps is motivated by the fact that the different instantiation steps are driven by different artifacts. Whereas the mapping between abstract modeling expressions and abstract web service methods is highly application driven, the mapping between the latter and the different framework APIs is mainly framework driven. Since the first mapping is application dependent, it is highly variant and usually changes from application to application. The latter mapping is framework dependent and then can be reused by another application if the same underlying framework technology is used.

Data instantiation and concrete values for executing the attack trace are given by configuration files. Examples of such data are: proxy IP address and port, WebGoat login credentials, URL of the WebGoat application, caption of buttons and links that have to be clicked, credentials of the user curly who was introduced in the model, criteria how the `moeProfile` is identified, etc. This list heavily depends on the model. If values in the model are set equally to the values in WebGoat, then these values do not have to be mapped. If abstract values are introduced in the model the configuration file
must specify the mapping between the abstract value in the model and the concrete value in WebGoat.

10.1.3 Results and discussion

We successfully reproduced the attack trace given by the model-checker in Listing 19. The model-checker was able to detect an attack because of the introduction of an authorization flaw inside the secure model. Thus it demonstrates that some authorization flaws can be detected by SPaClIoS tools. This authorization flaw is in the data layer but we also reproduced an attack due to an authorization flaw in the business layer (where someone without an administrator role deletes a profile).

We also tried to reproduce a stored cross-site scripting attack. However, our model does not distinguish a correct input validation from a flawed one. The model-checker gives then any trace from the model that contains an input and an output but cannot provide easily (nor efficiently) another trace. We are working closely with the model-checker developers to add the trace enumeration feature.

10.2 Assessment of test execution engine within SAML

We tested two SSO implementations based on SAML: the SAML-based SSO for Google Apps (hereafter, Google-SAML) and SimpleSAMLphp (hereafter, SimpleSAML). In both cases the objective was to check whether they suffered from the authentication flaw reported in [3]. In order to test these implementations, we reproduced the attack returned by the model checker (cf. [3]) using the approach to testing we developed during the first year of the project [22].

In the following we consider only the following components of the SPaClIoS Tool:

- **Model Checking**: It analyzes the model of the system looking for counterexamples (i.e. attack traces) that violate a security property.

- **Trace Improvement**: It extends the attack trace returned by the model checker in order to make it applicable to the actual system (i.e. the SUV).

- **Instrumentation**: It is responsible for enriching model with data and information useful for executing the test.
• **Test Execution Engine (TEE):** It manages the execution of the tests and their results. It drives the execution of the test by relying on the functionalities offered by the Test Driver.

• **Test Driver:** it provides API calls for building and parsing messages to/from the SUV. Moreover, it exchanges concrete messages with the SUV.

**Formal Modeling.** The first step is the definition of an ASLan++ specification (hereafter, the SAML model) which is available in Appendix C. This is done in a way that the attack traces returned by the model checker (if any) contain information useful to automate the execution of the test. To this end, we use function symbols to abstractly represent the structure of the messages exchanged among the principals. This simplifies the construction of concrete messages by the TEE. For instance, let us consider the following fragment of the specification, representing (part of) the transitions of C:

\[
\text{Actor} \rightarrow* \text{SP} : \text{httpReq(get, URI, nil\_h\_el, nil\_h\_el)}; \\
\text{SP} \rightarrow* \text{Actor} : \text{httpResp(code\_30x, IdP, ?AReq, ?Body)};
\]

The security analyst also specifies the correspondence between the formal model and the actual system. This is done by specifying a Refinement Mapping (RefMap). The RefMap we used for Google-SAML is as follows:

\[
\begin{align*}
\text{sp} &= \{ \text{"acs"} : \text{"https://www.google.com/a/ai-lab.it/acs"}, \\
& \quad \text{"entity"} : \text{"google.com"}, \\
& \quad \text{"url"} : \text{"http://mail.google.com/a/ai-lab.it/h"} \} \\
\text{i} &= \{ \text{"acs"} : \text{"https://malicious-sp.ai-lab.it/acs"}, \\
& \quad \text{"entity"} : \text{"mallory"}, \\
& \quad \text{"url"} : \text{"https://malicious-sp.ai-lab.it/resource"} \} \\
\text{idp} &= \{ \text{"acs"} : \text{"https://idp.ai-lab.it/acs"}, \\
& \quad \text{"entity"} : \text{"ai-lab.it"}, \\
& \quad \text{"url"} : \text{"https://idp.ai-lab.it/"} \} \\
\text{\ldots} \\
\tau &= \text{"Roberto Carbone"}
\end{align*}
\]

\[
\begin{align*}
\text{get} &= \text{"GET"} \\
\text{post} &= \text{"POST"} \\
\text{url}\_\text{sp} &= \text{"http://mail.google.com/a/ai-lab.it/h"} \\
\text{url}\_i &= \text{"https://malicious-sp.ai-lab.it/resource"} \\
200 &= \text{"200"}
\end{align*}
\]

\(^{23}\)In comparison to the actual SAML model, we compressed the name of function symbols that occur in the examples and in the MSC graphs to improve readability.
Model Checking. The SAML model is submitted to the model checker (SATMC). SATMC discovers the Abstract Attack Trace (AATr) of Figure 31 as witness of the violation of the client authentication property. Our preliminary tests indicate that the usage of function symbols to represent the abstract structure of messages does not affect the performance of the model checker.

Trace Refinement. This phase builds an Improved Attack Trace (IATr) by adding missing transitions to the AATr. In the Google-SAML case, the IATr differs from the corresponding AATr for two transitions related to the authentication between C and IdP (cf. the original Google-SAML AATr in Figure 31 and the corresponding IATr in Figure 32).
Instrumentation. The SPaClIoS Tool (and in particular the TEE) must be instructed on how to

1. generate messages for the SUV,
2. parse messages from the SUV, and
3. update the internal state of the principals simulated by the TEE.

During the Instrumentation phase, the model in annotated with this information. In particular, each component $a$ of a messages is annotated as $\ell_a.p_a$ where $p_a$ is a program fragment that has to be executed by the TEE in order to (1) compute the concrete value of $a$ (for messages delivered to SUV), or (2) check whether the actual message matches the expected pattern (for messages coming from the SUV). $\ell_a$ is a pointer to an internal location of the SPaClIoS Tool, where the corresponding value, obtained by executing $p_a$, is stored.

To illustrate, consider the following fragment of the annotated version of the model:

\begin{verbatim}
C ->* SP : (outgoing message) 
\ell_m_a.p_a: hReq(\ell_get_p_get: get, \ell_uri_sp_p_uri_sp: uri_sp, ...);

SP *-> C : (incoming message) 
\ell_m_i.p_i: hResp(\ell_code_30x_p_code_30x: code_30x, \ell_uri_idp_p_uri_idp: uri_idp, ...);
\end{verbatim}

The output of this step is an Instrumented Model (IMod) with references to program fragments and concrete values.
**Test Execution Engine and Test Driver.** The TEE is an automaton that is responsible for executing and managing the test on the SUV. In particular, it receives as input:

1. the RefMap, defining which principals of the AMod belongs to the SUV (the others must be simulated by the TEE),
2. the IMod, and
3. the IATr.

By using this information the TEE is able to execute the test by simulating the principals of the AMod that do not belong to the SUV and by executing the attack trace on the SUV.

The TEE uses the API of the Test Driver to build outgoing messages and analyze incoming ones. At the current state of development, the Test Driver is implemented in Python and supports the construction of HTTP messages as well as of SAML Assertions and Requests.

### 10.2.1 Test results and conclusions

We assessed the SPaCioS Tool by running tests against Google-SAML and SimpleSAML. In both experiments we considered $\text{SUV} = \{\text{IdP, SP}\}$. The SPaCioS Tool was able to reproduce the attack trace described in [3] against Google-SAML. It automatically executed the attack trace reporting that Google-SAML suffers from the authentication flaw reported in [3]. The same SUV was used to test SimpleSAML. However, in this case the execution of the attack halted due to a mismatch between a message observed and the expected one. By inspecting the messages exchanged, we verified that the failure is due to additional checks SimpleSAML performs on messages.

Our experiments showed that the SPaCioS Tool is able to verify automatically whether attack traces can be successfully executed against real systems. However, when the execution fails, the SPaCioS Tool reports an inconclusive verdict. In fact, it may happen that the $\text{SUV}$ reacts in a non-deterministic way. In order to exclude the non-determinism as the cause, conversations must be inspected manually.
11 Conclusions

In this deliverable, we have provided initial versions of descriptions of the application scenarios considered in SPaCioS. This information forms the basis for further analysis and testing activities. We have defined evaluation criteria that we will use to evaluate the technologies and tools developed in the project. We have also reported on preliminary assessments: the results are mainly positive, although the SPaCioS technologies and tools are not fully developed yet.

In the second year of the project, application scenarios will evolve and become more mature, we will formalize and analyze more problem cases, more test cases will be generated, and we will perform more testing activities to assess the security testing technologies and tools developed in other WPs.
A WebGoat WSDLScanning XML file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://localhost:8080/webgoat/services/WSDLScanning"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:impl="http://localhost:8080/webgoat/services/WSDLScanning"
    xmlns:intf="http://localhost:8080/webgoat/services/WSDLScanning"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/soap/wsd1/
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/soap/wsd1/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- WSDL created by Apache Axis version: 1.2
    Built on May 03, 2005 (02:20:24 EDT) -->
    <wsdl:message name="getCreditCardRequest">
        <wsdl:part name="id" type="xsd:int"/>
    </wsdl:message>
    <wsdl:message name="getCreditCardResponse">
        <wsdl:part name="getCreditCardReturn" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="getLoginCountRequest">
        <wsdl:part name="id" type="xsd:int"/>
    </wsdl:message>
    <wsdl:message name="getLoginCountResponse">
        <wsdl:part name="getLoginCountReturn" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="getFirstNameRequest">
        <wsdl:part name="id" type="xsd:int"/>
    </wsdl:message>
    <wsdl:message name="getFirstNameResponse">
        <wsdl:part name="getFirstNameReturn" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="getLastNameRequest">
        <wsdl:part name="id" type="xsd:int"/>
    </wsdl:message>
    <wsdl:message name="getLastNameResponse">
        <wsdl:part name="getLastNameReturn" type="xsd:string"/>
    </wsdl:message>
    <wsdl:portType name="WSDLScanning">
        <wsdl:operation name="getFirstName" parameterOrder="id">
            <wsdl:input message="impl:getFirstNameRequest" name="getFirstNameRequest"/>
            <wsdl:output message="impl:getFirstNameResponse" name="getFirstNameResponse"/>
        </wsdl:operation>
        <wsdl:operation name="getLastName" parameterOrder="id">
            <wsdl:input message="impl:getLastNameRequest" name="getLastNameRequest"/>
            <wsdl:output message="impl:getLastNameResponse" name="getLastNameResponse"/>
        </wsdl:operation>
        <wsdl:operation name="getCreditCard" parameterOrder="id">
            <wsdl:input message="impl:getCreditCardRequest" name="getCreditCardRequest"/>
            <wsdl:output message="impl:getCreditCardResponse" name="getCreditCardResponse"/>
        </wsdl:operation>
        <wsdl:operation name="getLoginCount" parameterOrder="id">
            <wsdl:input message="impl:getLoginCountRequest" name="getLoginCountRequest"/>
            <wsdl:output message="impl:getLoginCountResponse" name="getLoginCountResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="WSDLScanningSoapBinding" type="impl:WSDLScanning">
        <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="getFirstName">
            <wsdlsoap:operation soapAction="/"/>
            <wsdl:input name="getFirstNameRequest"/>
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```
namespace = "http://lessons.webgoat.owasp.org"
use = "encoded"/>
</wsdl:input>
</wsdl:operation>

<wsdl:operation name="getLastName">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="getLastNameRequest">
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://lessons.webgoat.owasp.org"
use="encoded"/>
</wsdl:input>
</wsdl:operation>

<wsdl:operation name="getCreditCard">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="getCreditCardRequest">
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://lessons.webgoat.owasp.org"
use="encoded"/>
</wsdl:input>
</wsdl:operation>

<wsdl:operation name="getLoginCount">
<wsdlsoap:operation soapAction=""/>
<wsdl:input name="getLoginCountRequest">
<wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
namespace="http://lessons.webgoat.owasp.org"
use="encoded"/>
</wsdl:input>
</wsdl:operation>

</wsdl:binding>
<wsdl:service name="WSDLScanningService">
<wsdlsoap:port binding="impl:WSDLScanningSoapBinding" name="WSDLScanning">
<wsdlsoap:address location="http://localhost:8080/webgoat/services/WSDLScanning"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
B WebGoat ASLan++ code

specification wg_rbac_1
channel_model CCM

definitions

entity Environment {

types
  profile < message;
  cookie < message;

symbols
  login(agent, symmetric_key): message;
  viewProfileOf(agent): message;
  nonpublic noninvertible password(agent, agent): symmetric_key;
  % used by Server entity to store users' cookies
  nonpublic cookies(agent): (agent * cookie) set;
  canView(agent, agent): fact;
  canVisit(agent, agent): fact;
  hasProfile(agent, profile): fact;
  % Specify who is part of the SUT
  inSUT(agent): fact;

clauses
  canViewOwnProfile(A, P): A-> canView(A) :- A-> hasProfile(P);
  canViewBProfile(A, B): A-> canVisit(B);

symbols
  % local variables for the lesson example
  webServer: agent;
  moe, curly: agent;
  nonpublic moeProfile, curlyProfile: profile;

entity Session(U, S: agent) {

entity User(Actor, S: agent) {
	symbols
    A: agent;
    Profile: profile;
    Cookie: cookie;

  body {
    % login
    Actor --> S : login(Actor, secret_pwd:(password(Actor, S)));
    S --> Actor : secret_cookie:(?Cookie); % user gets his cookie
    % viewProfile
    if (Actor-->canView(?A)) {
      Actor --> S : Cookie.viewProfileOf(A);
      S --> Actor : A.?Profile;
    }
  }
}

definitions

entity Server(U, Actor: agent) {
	symbols
    A: agent;
    Profile: profile;
    Cookie: cookie;

  body {
    while(true) {
      select { % the server acts upon receiving messages from a client
        % Login action
        on(?U --> Actor: login(?U, secret_pwd:(password(?U, Actor)))); {

        % viewProfile
        if (Actor-->canView(?A)) {
          Actor --> S : Cookie.viewProfileOf(A);
          S --> Actor : A.?Profile;
        }
      }
    }}}}
% create a new cookie for this user
secret_cookie : (Cookie) := fresh();
% assign created cookie to the user (U) and store this relation
cookies(Actor) => contains((U, Cookie));
Actor =>** U : Cookie; % send cookie back to user
%
% ViewProfile action
on (?U =>* Actor : ?Cookie.
  & cookies(Actor) => contains((?U, ?Cookie))
  & ?U => canView(?A) % Comment out this to exhibit an attack
): {
  % get the profile of A
  select{ on(A => hasProfile(?Profile)):{} }
  Actor =>** U : A.Profile;
}
%
%
body { % of Session
  new Server(U, S);
  new User(U, S);
}
goals
secret_pud:(_,) {U,S};
% secret_cookie:(_,) {U,S}; % the set of agents who may know the cookie
%
body { % of Environment
  %% agents that are part of the SUT
  inSUT(curly);
inSUT(moe);
inSUT(i);
  %% set profiles
  moe => hasProfile(moeProfile);
curly => hasProfile(curlyProfile);
  %% moe can visit curly’s profile
  moe => canVisit(curly);
  %% Let assume that curly has been compromised
  %% Then i can play curly’s role
  dishonest(curly);
  iknows(inv(ak(curly)));
  iknows(inv(ck(curly)));
  iknows(inv(pk(curly)));
  i => canVisit(curly);
  iknows(password(curly, webServer));
  iknows(curlyProfile);
  %% Create a session between anyone, including i, and the webServer
  any U. Session(U, webServer) where inSUT(U);
}
goals
secret_profiles:
  forall A P. [](iknows(P) & A => hasProfile(P) => i => canView(A));
secret_cookies:
  forall A C. [](iknows(C) & cookies(webServer) => contains((A, C)) => dishonest(A));

% For Emacs:
% Local Variables:
% tab-width: 2
% End:
C SAML ASLan++ code

```
specification SAML_SSO_SP_init_unil
channel_model CCM

entity Environment {
  types
  int < message;
  uri < agent;
  method < message;
  code < message;
  http_element < message;
  http_body < http_element;
  saml_message < message;
  saml_binding < http_element;

  symbols
  % % NIL values
  nil : message;
  nil_agent : agent;
  nil_http_element : http_element;

  % % HTTP protocol values
  get, post : method;
  code_30x, code_200 : code;
  uri_sp, uri_i : uri;

  % % Agents
  TrustedSPs : agent set;
  c, sp, idp : agent;

  % % Values for the intruder
  fake_id : int;

  % % Labels
  httpRequest(method, agent, http_element, http_element) : message;
  httpResponse(code, agent, http_element, http_element) : message;
  htmlForm(agent, saml_binding) : http_body;
  authnRequest(agent, agent, int) : saml_message;
  noninvertible
  signedAuthnResponse(private_key, agent, agent, agent, int) : saml_message;
  httpBinding(saml_message, uri) : saml_binding;
  postBinding(saml_message, uri) : saml_binding;

  clauses
  analysis_signedAuthnResponse(K, A1, A2, A3, N):
    iknows(signedAuthnResponse(K, A1, A2, A3, N)) & iknows(K);

  % %%%%%%%%%%%%%%% SESSION %%%%%%%%%%%%%%%%
  entity Session (C, IdP, SP: agent, TrustedSPs : agent set, URI : uri) {

  % %%%%%%%%%%%%%%% CLIENT %%%%%%%%%%%%%%%%
  entity Client (Actor, SP, IdP: agent, URI : uri) {
    symbols
    AReq : httpBinding(authnRequest(agent, agent, int), uri);
    ARsp : postBinding(signedAuthnResponse(inv(public_key),
                              agent, agent, agent, int), uri);
    Resource : http_element;
```
body { 

%% C-SP (1) 
\[\text{Actor}_\{\text{pk(Actor)}\} \rightarrow \text{SP} : \text{httpRequest} (\text{get}, \text{C_on_uri}:(\text{URI}), \text{nil_http_element}, \text{nil_http_element}); \]
\text{SP} \rightarrow \text{[Actor]}_{\{\text{pk(Actor)}\}} : \text{httpResponse} (\text{code}_30x, ?\text{IdP}, ?\text{AReq}, \text{nil_http_element});

%% C-IDP 
\text{Actor} \rightarrow \text{IdP} : \text{httpRequest} (\text{get}, \text{IdP}, \text{AReq}, \text{nil_http_element}); 
\text{IdP} \rightarrow \text{Actor} : \text{httpResponse} (\text{code}_200, \text{nil_agent}, \text{nil_http_element}, \text{htmlForm} (?\text{SP}, ?\text{ARsp}));

%% C-SP (2) 
\text{[Actor]}_{\{\text{pk(Actor)}\}} \rightarrow \text{SP} : \text{httpRequest} (\text{post}, \text{SP}, \text{nil_http_element}, \text{ARsp}); 
\text{SP} \rightarrow \text{[Actor]}_{\{\text{pk(Actor)}\}} : \text{httpResponse} (\text{code}_200, \text{nil_agent}, \text{nil_http_element}, ?\text{Resource});
}

Xxxxxxxxxxxxxxxxxidendity provider xxxxxxxxxxxxxxxxxx
entity IdentityProvider (\text{Actor}, \text{C}, \text{SP} : \text{agent}, \text{TrustedSPs} : \text{agent set}) { 
  symbols
  AnyC : \text{public_key};
  ID : \text{int};
  URI : \text{uri};

  body { 
    \text{C} \rightarrow \text{Actor} : \text{httpRequest} (\text{get}, \text{Actor}, \text{httpBinding} (\text{authnRequest} (?\text{SP}, \text{Actor}, ?\text{ID}), ?\text{URI}), \text{nil_http_element});
    
    \text{if} (\text{TrustedSPs} \rightarrow \text{contains} (\text{SP})) { 
      \text{Actor} \rightarrow \text{C} : \text{httpResponse} (\text{code}_200, \text{nil_agent}, \text{nil_http_element}, \text{htmlForm} (?\text{SP}, ?\text{ARsp}));
    }
  }
}

Xxxxxxxxxxxxxxxxxx service provider xxxxxxxxxxxxxxxxxx
entity ServiceProvider (\text{Actor}, \text{IdP}, \text{C} : \text{agent}, \text{URI} : \text{uri}) { 
  symbols
  AnyC : \text{public_key};
  AnyURI : \text{uri};
  ID : \text{int};
  AA : \text{message};
  ConsumedAAs : \text{message set};
  Resource : \text{http_element};

  body { 
    [?]_\{?\text{AnyC}\} \rightarrow \text{Actor} : \text{httpRequest} (\text{get}, \text{URI}, \text{nil_http_element}, \text{nil_http_element});
    ID := \text{fresh}();
    \text{Actor} \rightarrow [?]_\{?\text{AnyC}\} : \text{httpResponse} (\text{code}_30x, \text{IdP}, \text{httpBinding} (\text{authnRequest} (\text{Actor}, \text{IdP}, \text{ID}), \text{URI}), \text{nil_http_element});
    [?]_\{?\text{AnyC}\} \rightarrow \text{Actor} : \text{httpRequest} (\text{post}, \text{Actor}, \text{nil_http_element},
postBinding(signedAuthnResponse(inv(pk(IdP)), 
Actor, IdP, ?C, ID), C_on_uri:(URI)));

Resource := fresh();
Actor *->* [?]_AnyC : httpResponse ( code_200 , nil_agent , 
nil_http_element , Resource);

% assert finished: false;
}
}
body {
%% a new session is built
new Client(C, SP, IdP, URI);
new IdentityProvider(IdP, C, SP, TrustedSPs);
new ServiceProvider(SP, IdP, C, URI);
}
goals
C_on_uri:(_ C *-> SP;
%SP_on_resource:(_ SP *->* C;

}
body {
TrustedSPs := {sp, i};
%% Sessions
new Session(c, idp, i, TrustedSPs, uri_i ); % honest C talking to i(SP)
new Session(c, idp, sp, TrustedSPs, uri_sp); % honest IdP and SP
}
)
D Infobase ASLan++ code

```plaintext
specification InfobaseScene1_Safe
channel_model CCM

entity Environment {

definition

entity Session (User , LoginService: agent) {

    symbols
    % init constants for the environment body
    Peter, LoginS, Frontend, Repo1: agent;
    UserName, Password: text;

    % shared facts
    userAuth: fact;

    entity User (Actor, LoginService: agent, UserNameU, PasswordU: text) {

        symbols
        CookieU: message;
        RequestU, AnswerU: message;
        Frontend: agent;

        body {
            % user sends user and pass to the LoginService
            Actor ->* LoginService: UserNameU.secret_Password:(PasswordU);

            % the Frontend entity responds to the login request (after LS
            % forwards user and pass) with a cookie
            ?Frontend *->* Actor: ?CookieU;

            % the user creates a new request
            RequestU := fresh();

            % the user sends to the fronted the request and a cookie
            % that gives him the access to the repository
            Actor *->* Frontend: CookieU.RequestU;

            % the Frontend sends back to the user the answer of the repository
            Frontend *->* Actor: ?AnswerU;

            % for executability test
            assert reached_breakpoint1: false;
        }

    }

    entity User (Actor, LoginService: agent, UserNameU, PasswordU: text) {

        symbols
        CookieU: message;
        RequestU, AnswerU: message;
        Frontend: agent;

        body {
            % user sends user and pass to the LoginService
            Actor -> LoginService: UserNameU.secret_Password:(PasswordU);

            % the Frontend entity responds to the login request (after LS
            % forwards user and pass) with a cookie
            ?Frontend *->* Actor: ?CookieU;

            % the user creates a new request
            RequestU := fresh();

            % the user sends to the fronted the request and a cookie
            % that gives him the access to the repository
            Actor *->* Frontend: CookieU.RequestU;

            % the Frontend sends back to the user the answer of the repository
            Frontend *->* Actor: ?AnswerU;

            % for executability test
            assert reached_breakpoint1: false;
        }

    }

} % end of User entity

} % end of User entity
```

FP7-ICT-2009-5, ICT-2009.1.4  
Project No. 257876
% the online Login Service entity

entity LoginService(Actor, Frontend, User: agent) {

symbols
  UserNameL, PasswordL: text;
  RoleL: text;
  UserIPL: agent;
  LoginDataBase: (text, text, text) set;

body {
  % receives authentication data from the user
  % while(true) {
    select {
      % no matter on who is the sender of the data
      % this means that an intruder can easily
      % perform a brute force attack
      % checks if the data are available in the database
      % select are "faster" than if
      % with the query below, we extract the role of the user
      % that sends us the credential data
      LoginDataBase -> contains((?UserNameL, ?PasswordL, ?RoleL))): {
        % LS sends the role and the username to the Frontend entity
        Actor *->* Frontend: RoleL. UserNameL. UserIPL;
      }
    }
    % assert userAuth_test: !userAuth; ??
    %} % end while
  } % end body LS entity
} % end Login Service entity

% the online Frontend entity

entity Frontend(Actor, LoginService, Repository1: agent) {

symbols
  RoleF, UserNameF: text;
  CookieF: message;
  RequestF, AnswerF: message;
  UserIPF: agent;
  checkPermissions(message, message): fact;
  nonpublic createCookie(text, text): message;

body {
  % first the Frontend receives the user request by the LS
  % the it creates the cookie and sends it back to the user
  % the creation of the cookie
CookieF := createCookie(RoleF, UserNameF);

% Frontend uses the IP sended by LS to communicate the cookie to the correct user
Actor *->* UserIPF : CookieF;

% After receiving the cookie the user makes his first request

% Check if the user can do this request
select {
  on (checkPermissions(CookieF, RequestF)):
  {
    % If the user has the right credential, then the Frontend can
    % send the request to the repository that will send back the answer
    Actor *->* Repository1 : RequestF;
    Repository1 *->* Actor : ?AnswerF;
    % The answer of the repository is sent to the user
    Actor *->* UserIPF : AnswerF;
  }
  % Otherwise the user is a cheater...
}

} % Body closure Frontend

} % End of Login Service entity

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
% The Repository1 entity

entity Repository1(Actor, Frontend: agent) {

  symbols
  RequestR, AnswerR: message;
  elaborateRequest(message): message;

  body {
    Frontend *->* Actor: ?RequestR;
    % After receiving the request repository
    % Make the operation and reply with an answer
    AnswerR := elaborateRequest(RequestR);
    Actor *->* Frontend: AnswerR;
  }

} % Body closure

} % End of Login Service entity

body {
  % of Session

  % User and Login Service instantiation
  new User(Peter, LoginS, UserName, Password);
  new LoginService(LoginS, Frontend, Peter);
  new Frontend(Frontend, LoginS, Repo1);
  new Repository1(Repo1, Frontend);
}

goals
%the Password must be a secret between the user and the LoginS
secret_Password:(_){Peter, LoginS};

} % end of Session

body { % of the Environment entity

% just one session
any User LoginService. Session(User, LoginService)
  where !User=LoginService;

}
References


[22] SPaCioS. Deliverable 2.1.1: Analysis of the relevant concepts used in the case studies: applicable security concepts, security goals and attack behaviours, 2011.