SPaCIoS

Secure Provision and Consumption in the Internet of Services

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

Project No. 257876

www.spacios.eu

Deliverable D3.4
Bridge components for different levels of abstraction

Abstract

This deliverable implements two further conceptual results from D2.5.1 [4] and solves the problem of generating driver components to interface a System Under Validation (SUV) that bridges the different levels of abstraction for a specified set of security properties and a specified set of SUV models to perform property-based security testing as in D3.2 [5]. In addition, it provides means to interface with existing penetration testing tools to support model-conscious penetration testing as in D3.3 [6].

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *03.10.2013* Due on: *30.09.2013*
Editors: *TUM, UNIVR, ETH Zurich, INP and IeAT principal editors; UNIGE, SAP and Siemens secondary editors* Total pages: *40*

Project details

Start date: *October 01, 2010* Duration: *36 months*
Project Coordinator: *Luca Viganò*
Partners: UNIVR, ETH Zurich, INP, KIT/TUM, UNIGE, SAP, Siemens, IeAT

SEVENTH FRAMEWORK PROGRAMME

(this page intentionally left blank)

Contents

List of Figures

List of Acronyms

AAT Abstract Attack Trace . . . 14
ASLan++ high level AVANTSSAR Specification Language
DY Dolev-Yao
GA Generating Action . . . 18
IBT Instrumentation-Based Testing . . . 14
IUT Implementation Under Test . . . 14
SUT System Under Test . . . 6
SUV System Under Validation . . . 1
TEE Test Execution Engine . . . 14
VA Verifying Action . . . 18
WAAL Web Application Abstract Language . . . 3

1 Introduction

In D2.5.1 [4] we described the conceptual framework that allows one to concretize abstract tests generated by the SPaCIoS tool. In this deliverable, we give details on how these conceptual results are implemented in each of the SPaCIoS components, with an emphasis on design decisions that will be reflected on the SPaCIoS tool. Moreover, we discuss connections of the low-level vulnerability technology presented in D3.2 [5] and D3.3 [6] with other low-level tools that do not use models to guide their tests.

So-called *drivers* are needed both for the model inference and for the test case generation components. The role of a driver can be two-fold: it concretizes abstract messages from the model (in case of IBT and SPaCiTE) and it abstracts concrete messages from the System Under Test (SUT) (in case of model inference). This way, a relation between the abstract model and the SUT is established. Each section summarizes the experience on practically implementing the concepts presented in D2.5.1 [4] for the single tool components, and presents how the driver generation works for its specific context. It also describes the configuration steps that allow a user to tweak the driver generation for particular testing purposes.

This deliverable is organized as follows: first of all, Section 2 shows how test drivers are generated for the model inference tool components. Then, Section 3 describes the driver generation for each of the two test case generation tools present in SPaCIoS: IBT and SPaCiTE. Finally, Section 4 discusses the link that has been established between the Vera tool and existing penetration tools in order to make this technology *model-conscious*.

2 Generating test drivers for model inference

2.1 Introduction

Automata learning techniques currently enable the inference of Mealy Machines and subsets of Extended Finite State Machines.

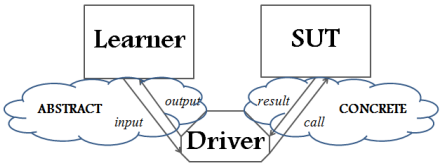

Figure 1: Learning setup

The learning setup used in these techniques is comprised of a System Under Test (SUT), a learner and a driver (see Figure 1). The SUT represents the black-box system to be learned. The learner sends inputs to the system under test and reads the resulting outputs. Based on the observed results, it infers a state model of the system through an iterative process. The driver component mediates communication between the SUT and the learner by calling functionality on the SUT as described by abstract inputs, and inferring the abstract outputs from the results.

2.2 Motivation and solution

To use a state-of-the-art learning tool, one has to establish mappings between abstract and concrete inputs/outputs and write the driver to implement those mappings. The driver lies between the learner and SUT, mediating communication between the two.

A driver depends on the communication medium the learner uses to send inputs and to receive outputs. As an example, Tomte [1] transmits its messages using sockets, while SIMPA does it through direct method calls.

A driver is also dependent on how the functionality on the SUT is called and responses are received. Communication in web applications is based on HTTP requests and responses.

Lastly, a driver also needs to implement the mapping mechanism between abstract and concrete inputs and outputs. In order to do this, drivers rely on known abstractions for concrete messages as abstract symbols, which are traditionally established manually for each system. This conceptual abstraction process depends only on the system under test, and not on the learners. Moreover, for systems implemented in the same technology similar

abstraction patterns can be used. This leads to the idea of automating this conceptual abstraction process.

The second automation step for automatic generation of the driver is generating the code corresponding to the communication with the SUT and the learner. Combined, these two steps enable automatic driver generation.

It is possible that learners contain their driver generation framework or that a separate tool to provide support is developed. We cover both cases here. First, we describe the driver generation support integrated with SIMPA, then we present SaLSA (**S**emi-automatic **L**earning **S**etup **A**rchitect), a more general tool for driver generation, that, in addition to SIMPA, can also work with Tomte and supports learning of Java systems.

2.3 Driver Generation Support For SIMPA

In Deliverable 2.2.1 [3], we have presented an abstraction for web applications and described a method to generate this abstraction automatically using a crawler. The abstraction is written in an XML file which contains the required information about the system, its inputs, outputs and their parameters. Of course, the XML file can be edited by the user, e.g., to remove some inputs or add some values to the parameters.

To generate the abstraction of the system under inference (SUI), the user sets up a configuration file for the SUI. The crawler will analyze and extract each link, form and page and finally translate them into inputs and outputs.

From this XML file, the model inference component is able to run the inference without any other information. The API provided by the component also allows any other component to read and use this file as a test driver.

The XML is divided in three parts described below and illustrated with a test driver for WebGoat:

- *System*
  This part defines the information to access the system: host, port, cookies and credentials if needed. As the cookie values may not yet be valid when the test driver is loaded, an updated value will be requested from the user at the loading step.

  Listing 1: Settings for the SUI

```
<settings>
  <target>WebGoat_Stored_XSS</target>
  <host>localhost</host>
  <port>8080</port>
  <limitSelector>#lesson_wrapper</limitSelector>
```

```
  <cookies>userid=BE48F7DCBA178BBE;</cookies>
  <basicAuthUser>guest</basicAuthUser>
  <basicAuthPass>guest</basicAuthPass>
</settings>
```

- *Inputs*
  This part is composed of several input elements. Each input is represented by an URL and some combination of parameters. Each of these combinations will be used during the inference process. The combinations should contain valid and invalid values. All these values will be tested during the inference and they will be used to generate better guards. The test driver can also be used for concretizing abstract traces, only the names of the parameters are needed.

Listing 2: Inputs for the SUI

```
<input address="http://localhost:8080/WebGoat/" method="POST"
    type="FORM"> <parameters> <parametersCombination> <!-- valid -->
    <parameter name="action">Login</parameter> <parameter
    name="employee_id">101</parameter> <parameter
    name="password">larry</parameter> </parametersCombination>
    <parametersCombination> <!-- invalid --> <parameter
    name="action">Login</parameter> <parameter
    name="employee_id">101</parameter> <parameter
    name="password">john</parameter> </parametersCombination>
    </parameters> </input>
```

- *Outputs*
  This part contains the source code for the output pages and the corresponding paths for the parameters. When the system sends a HTTP response, the page will be compared with all the pages already known and the corresponding abstract output will be selected.

Listing 3: Outputs for the SUI

```
<output>
  <source>...</source>
  <parameters>
    <parameter>0/1/1/0/0/6/1/</parameter>
    <parameter>0/1/1/0/0/5/3/</parameter>
    ...
  </parameters>
</output>
```

2.3.1 Use by third-party software

SIMPA provides a set of functions to read and use the generated test driver through the class named *GenericDriver*. It allows a user to send and receive abstract data. In this way, third party code could use the driver generation without using the inference part of SIMPA. In the following example, we show an example of Java code to execute an abstract login request and print the abstract response.

Listing 4: Calling the test driver

```
public class Test {
  public static void main(String[] args) throws Exception
  {
    GenericDriver driver =
          new GenericDriver("my_abstraction.xml");
    ParameterizedInput aninput =
      new ParameterizedInput("login");
    aninput.addParameters(new Parameter("login"));
    aninput.addParameters(new Parameter("111"));
    aninput.addParameters(new Parameter("john"));
    ParameterizedOutput anoutput = driver.execute(aninput);
    System.out.println(anoutput);
  }
}
```

When the *execute* method is called, the driver will search the input corresponding to the name *login* and an HTTP request is created and filled with the values given in the parameters. Then, this request is sent to the system and the answer will be compared to the set of existing pages to generate the abstract response. At the end, the parameters are extracted from the page.

2.4 SaLSA tool

SaLSA generates compilable drivers for applications implemented in a given technology, targeting a given learner. The pool of supported technologies are Java systems and HTTP web applications. In terms of Java systems, the tool supports RMI applications and also Java systems that are easily instantiated. We mean by this that setting up the system is done via a parameter-less **new** instantiation of a Java class implementing the system. The learners supported are Tomte and SIMPA.

The system architecture is centered around two concepts, the learner used and the technology in which the SUT is implemented. A scheme of the tool is given in Figure 2. For brevity, not all constituents have been included here.

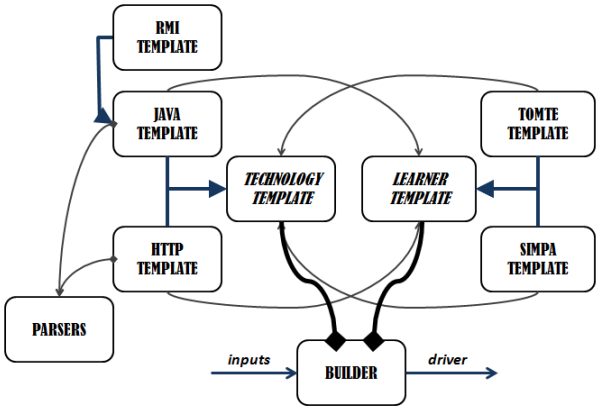

Figure 2: System architecture

The main components of the tool are an option reader, parsers specific to the system technology and learner and technology templates. The option reader fetches from a configuration file the following information: the technology in which the system is implemented, the learner for which generation is targeted and for each choice, it reads the specific settings for the selected learner/technology. For example, if RMI is selected, the tool will expect, among others, the path to the Java interface. If HTTP is selected, the tool will require a configuration file, describing the set of requests and responses used for learning. The settings available for each learner/technology can be configured via an options file. Each setting can be set as mandatory or optional, with a default value.

Also based on the selections, the tool loads the appropriate learner and technology templates. The technology template is connected then to the learner template and the resulting assembly is then called to generate the Java files implementing the driver.

Within technology templates, technology specific parsers are instantiated. These parsers are used to either infer or to read from a description, the abstract interface of the system, as a result of the proposed mapping presented in Deliverable 2.2.1 [3]. The interface object is then forwarded to the learner templates. The learner templates then build on this interface, generating the tool specific methods.

To facilitate extensibility, we have extracted abstract functionality that learner and technology templates should universally implement.

Thus, all technology templates should be capable of generating code for:

- **instantiating** the system or initiating the connection for web applications

- **resetting** the system
- **executing** on an input and fetching the output

Learner templates should be capable of generating code for:

- **accessing learner inputs** (input symbol and associated parameters)
- **creating learner outputs**, obtained from system outputs derived from executing the system
- **learner interaction**, i.e., receiving learner inputs from the learner and relaying learner outputs to the learner

2.5 Experimental results

We have tested SaLSA on the Stored XSS lesson from WebGoat. Our work was inspired by previous work done on learning WebGoat in [3].

For automated driver generation to be possible, along with means of describing actions and parameters used in the learning process, we have also added means of specifying request parameters that do not take part in the learning abstraction, but are introduced in the requests forwarded to the application. Such parameters can have fixed values (for example, 'name=John'), or they can have values that can be inferred from the corresponding output result. For example, if a response yields the content containing the string 'menu=900', the value of menu can be inferred from the content. This was needed, as there are cases when the parameter valuation can only be established at runtime.

```
technology http
learner simpa
systemport 8080
systemhost localhost
systempath /WebGoat/attack
descriptionfilepath webgoatdescfile.yaml
webgoatdescfile.yaml
```

The configuration file introduces, along with the learner and technology settings, the service hostname, the service port and path to the interface description file. The interface description file contains information assumed to be known by the web application(request parameters and possible values, discriminators and page descriptions), along with mechanisms for resetting and initiating the web application.

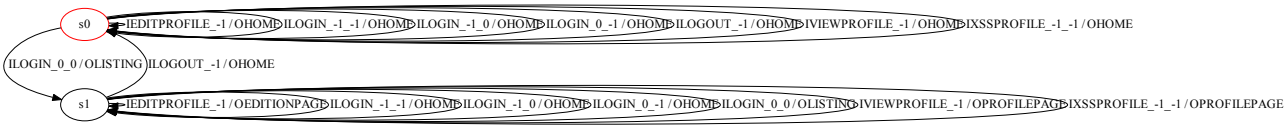

Figure 3: WebGoat stored XSS model learned by Tomte

The resulting model accurately describes the web application. A similar result is achieved with Tomte, the main difference being that strings are replaced with numbers in the representation, as Tomte only works with integer parameters. The mapping from integers to strings is handled by the generated driver. Since SIMPA was already shown to work, we depict the model as inferred by Tomte in Figure 3.

To summarize, we've implemented an assisted driver generation tool that supports both SIMPA and Tomte. We have tested drivers generated for each learner on a WebGoat lesson, receiving similar results, concluding that, both learners can be used with equal success in the learning of web applications.

3 Generating test drivers for test case execution

3.1 Test drivers for security protocols (IBT)

In [4, Section 3.1], we presented the overall concept how the Instrumentation-Based Testing (IBT) transforms Abstract Attack Traces (AATs) into test cases that can be executed by the Test Execution Engine (TEE). In a nutshell, IBT executes tests in two steps. First, it instruments a model with Java program fragments. Then it executes the Java fragments in the order established by the attack trace.

Besides the model, these two steps require as input an Implementation Under Test (IUT) and an adapter. Both these two inputs are core elements to make AATs concrete and executable. The adapter can also optionally include missing steps i.e., those steps of the real world that were abstracted away as not relevant for the specific security analysis that was performed. Here, we describe in details the IUT, the adapter, and the missing steps framework.

3.1.1 IUT

Figure 4 shows an excerpt of the IUT used for the SAML-based SSO for Google Apps (see [7, Section 3]). The IUT comprises the mapping between abstract and concrete symbols (see the panel on the top of Figure 4), as well as the list of protocol agents that are under test (see the panel on the bottom-right of Figure 4).

Let us discuss the mapping between abstract and concrete symbols. For instance, the abstract symbol `uri_sp` corresponds to the real world URL `http://www.google.com/calendar/hosted/ai-lab.it`. Similarly, the abstract symbol `httpRequest` is mapped to the adapter class `org.spacios.testdriver.Http`. That adapter class shall comprise a method to construct the real world representation of `httpRequest`, as well as methods to parse and process real world representations of `httpRequest`. This is explained further in Section 3.1.2. (The red highlighted objects just indicates that the adapter class is not yet implemented.) In the same line, the abstract symbol `userlogin` is mapped to the function `saml_sso.missingsteps.GcalendarXailabAuth` that should also be implemented as part of the overall adapter.

Let us discuss the definition of the SUT. With reference to Figure 4, the SUT comprises the honest service provider `sp` and the identity provider `idp`.

All the other protocol agents (including the DY intruder) are simulated by the instrumented model and testing engine.

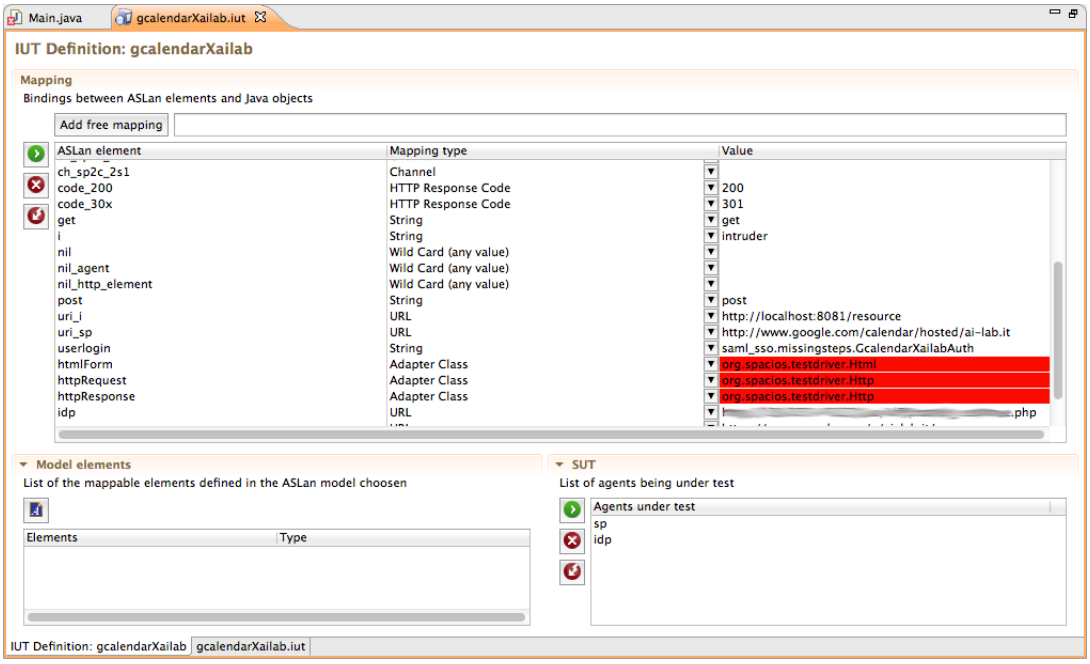

Figure 4: IUT for SAML-based SSO for Google Apps

3.1.2 Adapter

The adapter shall provide all the program fragments necessary to construct and process real world messages according to the mapping definition. In particular, for each abstract function, say $f(p_1, \ldots, p_n)$, whose function symbol f is mapped to an adapter class, then that class shall exist and shall implement the constructor method `constr_`f, as well as the decomposition methods `pi_1`,...,`pi_`n, i.e., one for each one of the parameters defining the abstract term. For instance, the abstract symbol `httpRequest` is abstractly defined in the ASLan++ specification of the SAML SSO as in Listing 5 and is mapped to the adapter class `org.spacios.testdriver.Http`. Therefore,

Listing 5: ASLan++ signature of `httpRequest`

```
httpRequest(method, agent, http_element, http_element)
```

the class `org.spacios.testdriver.Http` shall be created and implemented as in Listing 6.

Listing 6: Adapter's Java code fragment for `httpRequest`

```
package org.spacios.testdriver;

public class Http
{
  [..]
  public static HttpRequest constr_httpRequest(
       Object mArg, Object hpArg,
       Object uArg, Object bArg)
  {
    String method = (String) mArg;
    String dom_port = (String) hpArg;
    [..]
    HttpRequestBase req = httpMethodFactory(method, uri, body);
    return req;
  }

  public static String pi1_httpRequest(Object arg1)
  {
    HttpRequestBase hReq = (HttpRequestBase) arg1;
    return hReq.getMethod();
  }

  public static String pi2_httpRequest(Object arg1)
  {
    [..]
  }

  public static String pi3_httpRequest(Object arg1)
  {
   [..]
  }

  public static String pi4_httpRequest(Object arg1)
  {
   [..]
  }
```

It shall be noted that each one of the decomposition methods takes an input only one parameter and specifically a real world value of the object to be decomposed. While this is true for most of the abstract functions, there are some exceptions. Interpreted abstract symbol for cryptography such as `encrypt` and more in general all abstract symbols that are defined as non-invertible in the ASLan++ specification shall be handled in a specific way for what concerns the decomposition operation. In fact, their real world representation cannot be correctly decomposed without an additional parameter e.g., the proper private key to decrypt an encrypted element. The mapping allows the user to specify these cases that shall then be properly handled in the adapter. For instance, the `encrypt` abstract symbol is always associated to the mapping type "Adapter Class - Asymmetric Encryption" and will require the implementation in the class adapter of the method `asymDecription_encrypt` assumed to take an input two parameters, one being the encrypted element and one being the public key to decrypt it. This

will be properly documented and the SPaCIoS Tool will provide support to the users to be sure mapping and adapter get well together. For instance, if `encrypt` is mapped to an adapter class and this class does not implement the proper decomposition method, then the user will be alerted.

3.1.3 Missing steps framework

The missing steps framework is a simple way to deal with message exchanges that are not in the model. This framework is used for three distinct operations:

Abstract fact concretization: to detail an abstract fact in the model. In ASLan++ it is possible to assert any abstract fact representing the execution of certain actions in the real world. If such a fact appears during the execution of the AAT and represents some important operation to perform, then the user has to define the behavior of the testing platform for that fact. For instance, this is the case for the abstract fact `userlogin` of the SAML SSO specification. That fact represents the real world action of the SAML Client authenticating into the SAML Identity Provider and the adapter function `saml_sso.missingsteps.GcalendarXailabAuth` shall code how to handle this operation in the real world.

Deviation handling: to handle the deviations of the SUT from the model. Real implementations may present some discrepancies with their reference protocol model. There are often more messages exchanges in the real world protocol than those strictly captured in the formal definition of a protocol. To tackle this kind of events, we get advantage of an error handler based on missing steps. That error handler will define the actions to perform when the TEE receives a message which is not expected in the AAT. In such a case, the TEE will ask for a relevant `MissingStep` to the error handler, execute it (if any), and get back in the normal execution of the AAT.

Final steps: to perform some final actions after the execution of the AAT. Once the last expected message of the AAT is received and because, once again, the implementation can differ from the model, the user may want to continue beyond the AAT in order to get additional insight. For instance, let us consider a protocol that is authenticating two parties and providing a common shared key to the two of them. Of course, the shared key should not be accessible to an intruder. Let us imagine, however, that an AAT is discovered to violate that property. Executing

the AAT through the TEE will end with the intruder within TEE getting the shared key. The user may be interested to execute few final steps to exploit that shared key in order to test whether also protected resources under that key can be retrieved from the SUT.

The usage of the missing steps framework will be further detailed in the tutorials of the SPaCIoS Tool.

3.2 Test drivers for Web applications (SPaCiTE)

In [4, Section 3.2], we presented the overall concept how SPaCiTE instantiates AATs to test cases that can be executed by the TEE. As a short summary the TEE consists of a GUI application that allows the Security Analyst to load executable test cases and execute them. Because the TEE provides an abstract class of such a test case, the TEE is implemented in a generic way and is not test case specific. Test cases are loaded at runtime with the help of class loaders. Executable test cases are generated with a 2-step mapping that involves an intermediate language, called WAAL. Here, we describe in details the WAAL language. We also provide examples on how AATs are translated into WAAL and how WAAL actions are translated into Java source code (based on Selenium).

3.2.1 Web Application Abstract Language (WAAL)

The WAAL language is composed of two types of actions: Generating Action (GA) and Verifying Action (VA). The former aims at describing how to generate an abstract message by executing actions in a browser and the latter how to verify an abstract message. Listing 7 and Listing 8 list all available GAs and VAs, respectively.

In addition to specify how to generate and verify each abstract message of an AAT, the Security Analyst must also describe how to initialize the TEE and each simulated agent. Listing 9 lists the available functions for this purpose.

Finally, some mutation operators may require that the TEE injects a malicious content at a specific moment. This is indicates by the presence of the tags `INJECT` and `VERIFY` in the AAT. The former stipulates what to inject and the latter informs the TEE where to verify the action of the malicious content. Thus, the WAAL language contains also two extra actions for this purpose, as described in Listing 10. The `INJECT` and `VERIFY` tags have a strong correlation since the verification of the malicious effect strongly depends on the payload used during the injection. Therefore the `VERIFY`

Listing 7: Generation Actions

```
// Go to an URL
ga_goToURL(agent, url)
   agent: agent who performs the action
   url: text

// Press a button
ga_clickButton(agent, selector)
   agent: agent who performs the action
   selector: ByXPath(xpath expression) or ByText("button label")

// Click on a link
ga_clickLink(agent, selector)
   agent: agent who performs the action
   selector: ByXPath(xpath expression) or ByText("link caption")

// Select item(s) from a list
ga_selectItems(agent, selector, {elt1,elt2})
   agent: agent who performs the action
   selector: ByName("name of the select element")
   {elt1, elt2}: list of elements that are selected

// Type text into an input element
ga_inputText(agent, selector, text)
   agent: agent who performs the action
   selector: ByName("name of the input element")
   text: text to type in the input element
```

Listing 8: Verification Actions

```
// Verify if selector is present in the HTML page
va_HTMLPage(agent, selector)
   agent: agent who performs the action
   selector: ByXPath(xpath expression) or ByText("text")

// Verify if text is present in the section of the HTTPRequest
va_HTTPRequest(agent, section, text)
   agent: agent who performs the action
   section: HTTPRequest header or "body" where to look for
   text: text to look for

// Verify if text is present in the section of the HTTPResponse
va_HTTPResponse(agent, section, text)
   agent: agent who performs the action
   section: HTTPResponse header or "body" where to look for
   text: text to look for

// Verify if test is present in an Alert Window
va_Alert(agent, text)
   agent: agent who performs the action
   text: text to look for
```

action is determined automatically when choosing the `INJECT` action. That means that if the injection is about ReflectedXSS, the verification action verifies the same type of vulnerabilities. The payload type of the `INJECT` action is used to query the instantiation library to retrieve a Java object that implements both the injection and verification action. Invoking the injection method on this Java object returns a concrete, executable payload. Later on during the AAT the `VERIFY` action invokes the verification method on the same Java object that was used for the injection.

3.2.2 Mapping from abstract message to WAAL actions

The first mapping, provided by the user, maps each step of an AAT to a sequence of WAAL actions. Appendix A gives an example of such a mapping.

3.2.3 Mapping from WAAL actions to Java

The second mapping, which is application independent and therefore does not require any more information from the user, maps each WAAL action to Java statements such that the TEE can execute it on the SUT. Appendix B gives an example of a generated Java file.

Listing 9: Initialization Actions

```
init_tee()

shutdown_tee()

init_agent(name)
   name: agent name

// Connect to the selenium server playing this agent
config_tee_agent(agent, ip, port)
   agent: agent who performs the action
   ip: string for the IP address
   port: integer for the controller port

// Provide credentials for HTTP Basic Authentication
config_httpBasicAuthentication(agent, usr, pwd)
   agent: agent who performs the action
   usr: username
   pwd: password

generating_message(agent, desc) { GA-action* }
   agent: agent who generates the abstract message
   desc: description of the message

verifying_message(agent, desc) { VA-action* }
   agent: agent who verifies the abstract message
   desc: description of the message
```

Listing 10: Injection Actions

```
// Inject some malicious data
INJECT(agent, type, id, payload)
   agent: agent who performs the action
   type: SelectItems, InputText, or Unknown
   id: name of the input parameter
   payload: type of the payload (e.g., ReflectedXSS,
   ReflectedXSSSplitted, StoredXSS, StoredXSSSplitted,
   ReflectedSQL, ReflectedSQLSplitted, StoredSQL)

// Verify the result of the injected malicious data
VERIFY(agent)
   agent: agent who can performs the action
```

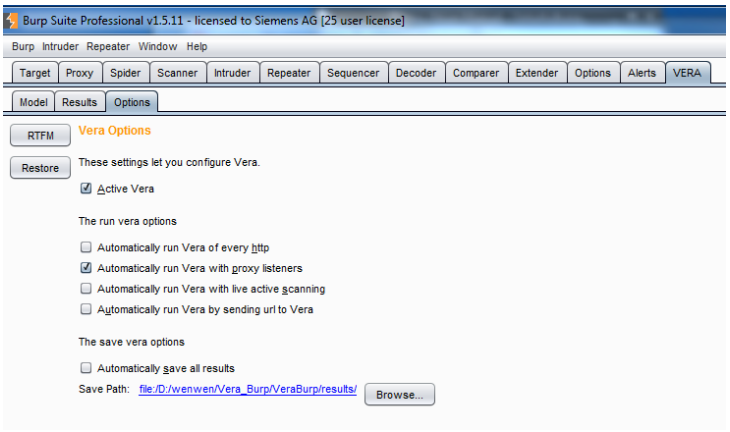

Figure 5: Preferences of the VERA plug-in for Burp.

4 Model-conscious penetration testing

The following section will focus on the possibility of using the Vera-tool with other tools in order to increase its potentiality.

As introduced in [6] the Vera-tool is used in order to create low-level attacker models and test these models against a SUT. The modularity of the python engine that performs the test inside the Vera-tool makes it well suited for its use with other tools. In the following we give two examples that will use the Command line interface of the Vera-tool in order to interface it with other tools.

4.1 Vera and Burp

The Burp Suite[1] is a security tool developed to aid testing web applications. It contains a number of different utilities ranging from proxy functionality to analysis of the quality of session ids. The professional version features an API that can be used to add extensions to the Burp tool.

We have developed a VERA plug-in for Burp which allows to run the low-level attacker models from within the Burp Suite, for example during Burp's automated scanning routine, therefore allowing a security analyst to merge the capabilities of both tools, or using the traffic from the proxy (see Figure 5). In this case, the VERA plug-in monitors all traffic that gets passed through the Burp proxy, and then proceeds to run the selected models using the traffic as the foundation for the attacks. This allows a security analyst

[1] http://www.portswigger.net

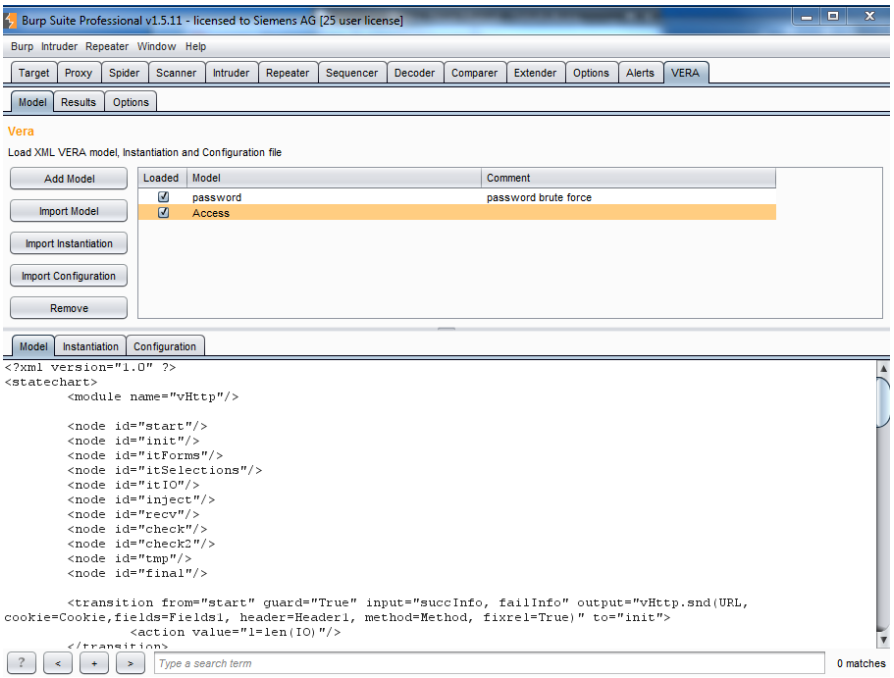

Figure 6: Model selection in the VERA Burp plug-in.

to go through complex work-flows in the web applications in order to test all the functionality using VERA.

The plug-in presents the user with an integrated user interface within Burp, see Figure 6, that allows the selection of low-level attacker models, instantiation libraries and configuration files as well as permitting minor changes.

As can be seen in Figure 7, the plug-in presents the security analyst with a view of all tests performed by the VERA plug-in, grouped according to the domain and the page.

4.2 Vera and Scrapy

Scrapy[2] is a fast high-level screen scraping and web crawling framework, used to crawl websites and extract structured data from their pages. It can be used for a wide range of purposes, from data mining to monitoring and automated testing.

We assume to be in a scenario where we have to validate a Cross Site Scripting attack on a web site. In figure 8 we report the Vera-model for a

[2] www.scrapy.org

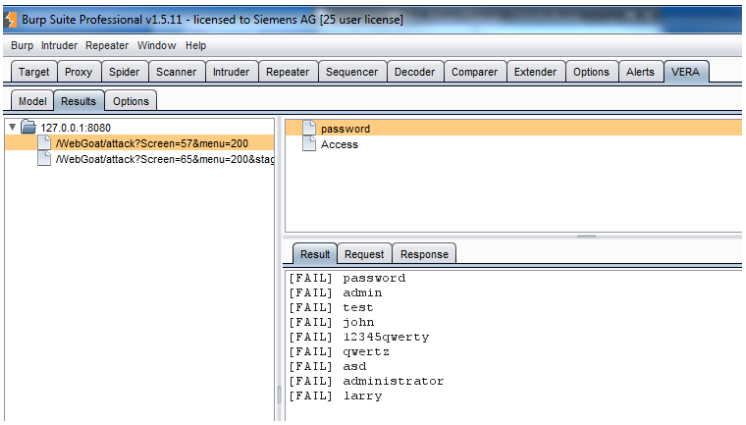

Figure 7: Results tab in the VERA Burp plug-in.

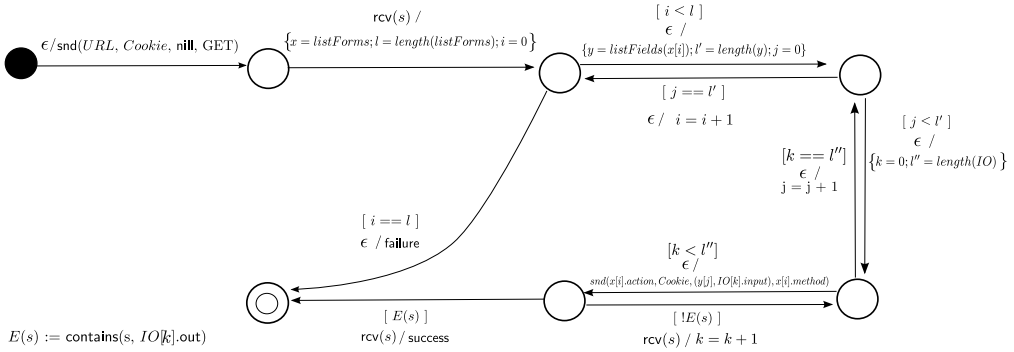

Figure 8: Vera-model for a general injection. The model can be used for injecting payloads in web page fields.

general injection that we will use along with a file containing payloads for a Cross Site Scripting attack.

The attacker formalized in the model of figure 8 follows a simple behavior:

- find all the forms on a page,
- select every field of each form, and
- inject the payloads in every fields.

In case we want to test every page of the target SUT the model must be redesigned, and its complexity will raise considerably. The solution for this problem can be found in the use of Scrapy for the crawling phase and the Vera-tool for the testing phase.

The following code can be used in Scrapy in order to extract php links matching ".php" (but not matching "example.php") and following these links inside the boundaries of a test domain.

```
from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor
from scrapy.selector import HtmlXPathSelector
from scrapy.item import Item

class TEST(CrawlSpider):
  name = 'TEST'
  allowed_domains = ['testDomain.org']
  start_urls = ['testDomain/index.php']

  rules = (
   Rule(SgmlLinkExtractor(allow=('\.php', ),
                          deny=('example\.php', ))),
   Rule(SgmlLinkExtractor(allow=('\.php', )),
                          callback='parse_item'),
   )

   def parse_item(self, response):
     hxs = HtmlXPathSelector(response)
     item = Item()
     return item
```

With the capability of crawling the pages of the SUT we can now use the following script against the SUT:

```
scrapy crawl TEST &> URLs.txt |
grep '(200)' URLs.txt | cut -d"<" -f2 |
                        cut -d" " -f2 | cut -d">" -f1 |
while read line; do
cp orig.conf $line;
echo "URL='$line'" >> $line;
./vera.py models/injection.xml libs/XSS-payloads $line;
done
```

In this example (and also in the following) the configuration file (orig.conf) contains all the parameters except for URL.

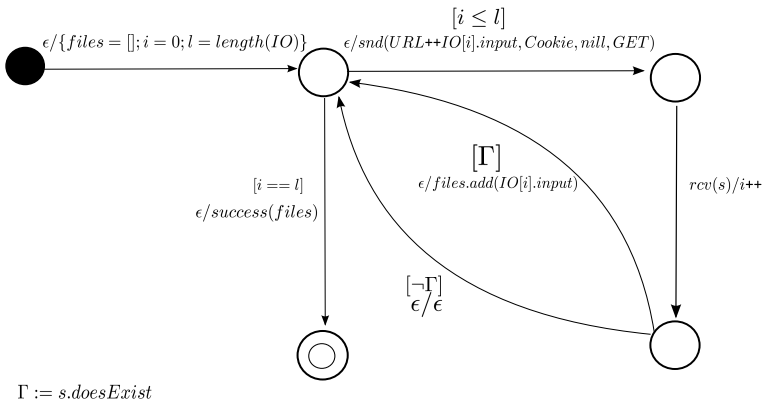

Figure 9: Vera-model for a file enumeration attack. The model can be used for trying to enumerate the file from a target URL

4.3 Vera and nmap

nmap[3] is a port scanning tool, which is used to gather information on the target of the test. Specifically, nmap, like all port scanners, attempt to locate which network services are available on each target host. They do this by probing each of the designated (or default) network ports or services on the target system.

In this scenario we use the model of a file enumeration attack shown in Figure 9. In this model the attacker

- tries to retrieve from the target URL each value in the instantiation library IO (appended to the URL),
- Receives the response from the SUT, and
- if the file exists he saves the URL.

We can use nmap and Vera in order to try to enumerate all the files in a network of server hosting the Hypertext Transfer Protocol (port 80) with the following script:

```
nmap -p80 <network> -oG - |
grep open |
cut -d\ -f2 |
while read line; do
cp orig.conf $line;
```

[3] www.nmap.org

```
echo "URL='http://$line/'" >> $line;
./vera.py models/file_enum.xml libs/file_enum $line;
done
```

In the example:

- `< network >` is the target range of IPs of the scan,
- -p80 searches for servers that are hosting Hypertext Transfer Protocol, and
- -oG makes nmap produces an output in grepable format.

5 Summary

In this deliverable, we address the problem of generating driver components to interface a SUT, both for model inference and for test case execution. Additionally, we provide a bridge to existing penetration testing tools by using existing black-box technology as the basis for a model-based approach with VERA (in the sense of D3.3 [6]). The design decisions presented in this deliverable serve as the basis for the implementation of the SPaCIoS tool presented in D4.2 [2].

For model inference, a driver depends on the communication medium the learner uses to send inputs and to receive outputs. A driver is also dependent on how SUT functions are called and how responses are received. For example, communication in Web applications is based on HTTP requests and responses, which constitute the basis for further abstraction.

Test case execution may reuse the knowledge acquired via model inference to concretize the abstract inputs and abstract the concrete outputs, in order to execute a test case and establish a verdict. Depending on which level of abstraction is used, different drivers are required. Thus, IBT relies on adapters for HTTP requests and responses while SPaCiTE relies on actions performed on a browser and formalized in the WAAL language.

Finally, we show how the technology developed in the context of D3.3 [6] can be linked with existing tools in order to achieve synergy between attack patterns as defined in VERA and readily available penetration testing technology. On the one hand, this connection can facilitate the adoption of the low-level attacker modeling technique proposed in the project by practitioners who rely on popular tools for certain routine tasks. On the other hand, this can reduce the complexity of models, where some information can be gathered with ad-hoc tools, and the core logic of the attack pattern is modeled with VERA.

References

[1] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. W. Vaandrager. Automata learning through counterexample guided abstraction refinement. In *18th International Symposium on Formal Methods (FM)*, volume 7436 of *LNCS*, pages 10–27. Springer, 2012.

[2] SPaCIoS. Deliverable 4.2: SPaCIoS Tool v.1 and Validation methodology patterns (final version), 2012.

[3] SPaCIoS. Deliverable 2.2.1: Method for assessing and retrieving models, 2013.

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

[5] SPaCIoS. Deliverable 3.2: SPaCIoS Methodology and technology for property-driven security testing, 2013.

[6] SPaCIoS. Deliverable 3.3: SPaCIoS Methodology and technology for vulnerability-driven security testing, 2013.

[7] SPaCIoS. Deliverable 5.3: Final Proof of Concept, 2013.

A WebGoat UDEF file

```
# User-defined mapping from abstract action to WAAL
# capitalize function (extracted from examples/capitalize2.m4)
define(‘_arg1’, ‘$1’)
define(‘_to_alt’, ‘changequote(‘<<[’, ‘]>>’)’)
define(‘_from_alt’, ‘changequote(<<[‘]>>, <<[’]>>)’)
define(‘_upcase_alt’, ‘translit(<<[$*]>>, <<[a-z]>>, <<[A-Z]>>)’)
define(‘_downcase_alt’, ‘translit(<<[$*]>>, <<[A-Z]>>, <<[a-z]>>)’)
define(‘_capitalize_alt’,
  ‘regexp(<<[$1]>>, <<[^\(\w\)\(\w*\)]>>,
    <<[_upcase_alt(<<[<<[\1]>>]>>)_downcase_alt(<<[<<[\2]>>]>>)]>>)’)
define(‘capitalize’,
  ‘_arg1(_to_alt()patsubst(<<[<<[$*]>>]>>, <<[\w+]>>,
    _from_alt()‘]>>_$0_alt(<<[\&]>>)<<[’_to_alt())_from_alt())’)
# Hint to make use of ‘ and ’ characters within macro definitions
define(‘LQ’, ‘changequote(<,>)‘dnl’
changequote‘’’)
define(‘RQ’, ‘changequote(<,>)dnl‘
’changequote‘’’)

define(‘getAgentIP’, ‘ifelse($1,tom,127.0.0.1,$1,jerry,127.0.0.1,
                          $1,david,127.0.0.1,UNKNOWN_AGENT_NAME)’)
define(‘getAgentPort’, ‘ifelse($1,tom,23007,$1,jerry,23009,
                          $1,david,23011,UNKNOWN_AGENT_NAME)’)
# Configure each simulated agent
define(‘CONFIGURE_AGENT’,
‘config_tee_agent($1, "getAgentIP($1)", getAgentPort($1))
config_httpBasicAuthentication($1, "guest", "guest")
’)

# Each simulated agent goes to the home page
define(‘START_AGENT’,
‘ga_goToURL($1, "http://172.16.1.11/WebGoat/attack")
ga_clickButton($1, ByText("Start WebGoat"))
ga_clickLink($1, ByText("Access Control Flaws"))
ga_clickLink($1, ByText("Stage 3: Bypass Data Layer Access Control"))
ga_clickLink($1, ByText("Restart this Lesson"))
’)

# Helper functions
define(‘employeeName’, ‘ifelse($1,tom,regexp:Tom.*,$1,
                          jerry,regexp:Jerry.*,
                          david,regexp:David.*,
                          UNKNOWN_EMPLOYEE_NAME)’)
define(‘password’, ‘ifelse($1,tom,tom,$1,jerry,jerry,
                          $1,david,david,UNKNOWN_PASSWORD)’)
define(‘listStaffOfXPATH’, ‘//span[@class=RQ()lesson_text_db‘’RQ()
                            and contains(.,RQ()capitalize($1)‘’RQ())]’)
```

```
define(‘getSSN’, ‘ifelse($1,jerry,858-55-4452,$1,tom,792-14-6364,
                         $1,david,439-20-9405,UNKNOWN_SSN)’)

# login
define(‘GENERATE_login’,
‘ga_selectItems($1, ByName("employee_id"), {"employeeName($2)"})
ga_inputText($1, ByName("pass‘’word"), "$3")
ga_clickButton($1, ByText("Login"))
’)

# viewProfileOf
define(‘GENERATE_viewProf’,
‘ga_selectItems($1, ByName("employee_id"), {"employeeName($2)"})
ga_clickButton($1, ByText("ViewProfile"))
’)

# editProfileOf
define(‘GENERATE_editProf’,
‘ga_clickButton($1, ByText("EditProfile"))
ga_inputText($1, ByName("address1"), "$3")
ga_clickButton($1, ByText("UpdateProfile"))
’)

# deleteProfileOf
define(‘GENERATE_deleteProf’,
‘ga_selectItems($1, ByName("employee_id"), {"employeeName($2)"})
ga_clickButton($1, ByText("DeleteProfile"))
’)

# listStaffOf
define(‘VERIFY_listStaff’,
‘va_HTMLPage($1, ByXPath("listStaffOfXPATH($2)"))
’)

# profileOf
define(‘VERIFY_profile’,
‘va_HTMLPage($1, ByText("getSSN($2)"))
’)
```

B Java file for an AAT in WebGoat

```
import javax.swing.*;
import java.lang.String;
import java.util.LinkedList;
import java.util.List;
import testcases.IExecutableAttackTrace;
import components.*;
```

```
public class webgoat_s1 extends IExecutableAttackTrace {
 long targetTime;
 boolean found;
 public static boolean successful = true;
 String displayText = null;
 Connections current_connections = null;

 public webgoat_s1(MainWindow win,
  JTextArea abstractArea,
  JTextArea waalArea,
  JTextArea httpArea,
  Boolean terminate) {
    super(win, abstractArea, waalArea, httpArea, terminate);
  }

  public void run() {

    List<WTCommand> _commands;
    // begin of a new step
    current_connections = getConnectionPool()
      .createConnectionsFor(
         "tom",
         "localhost",
         23006,
         "http://www.google.com",
         "127.0.0.1",
         23007);
    // disable proxy
    current_connections.InitizializeBarrier();
    current_connections.utils.finishStep(
       win,
       abstractArea,
       waalArea,
       httpArea );
    // end of the step

    // begin of a new step
    current_connections = getConnectionPool()
      .getConnectionsFor("tom");
    current_connections
      .setBasicAuthenticationCredentials("guest", "guest");
    current_connections.utils
      .finishStep( win, abstractArea, waalArea, httpArea );
```

```
// end of the step

// begin of a new step
current_connections = getConnectionPool()
  .createConnectionsFor(
     "jerry",
     "localhost",
     23008,
     "http://www.google.com",
     "127.0.0.1",
     23009);
// disable proxy
current_connections.InitizializeBarrier();
current_connections.utils
  .finishStep( win, abstractArea, waalArea, httpArea );
// end of the step

// begin of a new step
current_connections = getConnectionPool()
  .getConnectionsFor("jerry");
current_connections
  .setBasicAuthenticationCredentials("guest", "guest");
current_connections.utils
  .finishStep( win, abstractArea, waalArea, httpArea );
// end of the step

// begin of a new generation step
displayText = "Homing␣sequence" ;
// get connections for 'tom'
current_connections = getConnectionPool()
  .getConnectionsFor("tom");
current_connections.utils.prepareAction(
    win,
    displayText,
    current_connections,
    abstractArea );
_commands = new LinkedList<WTCommand>();
current_connections = getConnectionPool()
  .getConnectionsFor("tom");
current_connections.selenium
  .open("http://172.16.1.11/WebGoat/attack");
_commands.add( new ClickButtonHTML(
    win,
```

```
        current_connections,
        "//input[@value='Start␣WebGoat']") );
    _commands.add( new ClickLinkHTML(
        win,
        current_connections,
        "link=Access␣Control␣Flaws") );
    _commands.add( new ClickLinkHTML(
        win,
        current_connections,
        "link=Stage␣3:␣Bypass␣Data␣Layer␣Access␣Control") );
    _commands.add( new ClickLinkHTML(
        win,
        current_connections,
        "link=Restart␣this␣Lesson") );
    while( true ) {
      try {
        current_connections.utils.executeWithOptions(
            win,
            current_connections,
            _commands );
        break;
      } catch (ExceptionNoMoreExamplesAvailable e) {
        // ask test expert
        // it has to provide manual http request information
        current_connections.utils
          .handleExeptionAskTestExpert(current_connections);
        break;
      } catch (ExceptionAskTestExpert e) {
        // ask test expert
        current_connections.utils
          .handleExeptionAskTestExpert(current_connections);
        break;
      }
    }
    current_connections.utils
      .finishGeneratingAction( win, current_connections );
    current_connections.utils
      .finishStep( win, abstractArea, waalArea, httpArea );
    // end of the step

    // begin of a new generation step
    displayText = "Homing␣sequence" ;
    // get connections for 'tom'
```

```
current_connections = getConnectionPool()
  .getConnectionsFor("jerry");
current_connections.utils.prepareAction(
    win,
    displayText,
    current_connections,
    abstractArea );
_commands = new LinkedList<WTCommand>();
current_connections = getConnectionPool()
  .getConnectionsFor("jerry");
current_connections.selenium
  .open("http://172.16.1.11/WebGoat/attack");
_commands.add( new ClickButtonHTML(
    win,
    current_connections,
    "//input[@value='Start␣WebGoat']") );
_commands.add( new ClickLinkHTML(win,
    current_connections,
    "link=Access␣Control␣Flaws") );
_commands.add( new ClickLinkHTML(
    win,
    current_connections,
    "link=Stage␣3:␣Bypass␣Data␣Layer␣Access␣Control") );
_commands.add( new ClickLinkHTML(
    win,
    current_connections,
    "link=Restart␣this␣Lesson") );
while( true ) {
  try {
    current_connections.utils.executeWithOptions(
        win,
        current_connections,
        _commands );
    break;
  } catch (ExceptionNoMoreExamplesAvailable e) {
    // ask test expert
    // it has to provide manual http request information
    current_connections.utils
      .handleExeptionAskTestExpert(current_connections);
    break;
  } catch (ExceptionAskTestExpert e) {
    // ask test expert
    current_connections.utils
```

```
        .handleExeptionAskTestExpert(current_connections);
      break;
    }
  }
  current_connections.utils.finishGeneratingAction(
      win,
      current_connections );
  current_connections.utils.finishStep(
      win,
      abstractArea,
      waalArea,
      httpArea );
  // end of the step

  // begin of a new generation step
  displayText = "login(jerry,password(jerry,webServer))" ;
  // get connections for 'tom'
  current_connections = getConnectionPool()
    .getConnectionsFor("jerry");
  current_connections.utils.prepareAction(
      win,
      displayText,
      current_connections,
      abstractArea );
  _commands = new LinkedList<WTCommand>();
  _commands.add( new SelectItemsHTML(
      win,
      current_connections,
      "//select[@name='employee_id']",
      new String[]{"regexp:Jerry.*"}) );
  _commands.add( new InputTextHTML(
      win,
      current_connections,
      "//input[@name='password']",
      "jerry") );
  _commands.add( new ClickButtonHTML(
      win,
      current_connections,
      "//input[@value='Login']") );
  while( true ) {
    try {
      current_connections.utils.executeWithOptions(
          win,
```

```
            current_connections,
            _commands );
        break;
      } catch (ExceptionNoMoreExamplesAvailable e) {
        // ask test expert
        // it has to provide manual http request information
        current_connections.utils
          .handleExeptionAskTestExpert(current_connections);
        break;
      } catch (ExceptionAskTestExpert e) {
        // ask test expert
        current_connections.utils
          .handleExeptionAskTestExpert(current_connections);
        break;
      }
    }
    current_connections.utils.finishGeneratingAction(
        win,
        current_connections );
    current_connections.utils.finishStep(
        win,
        abstractArea,
        waalArea,
        httpArea );  // print separation lines
    // end of the step

    // begin of a new verification step
    displayText = "listStaff(jerry)" ;
    // get connections for 'tom'
    current_connections = getConnectionPool()
      .getConnectionsFor("jerry");
    current_connections.utils.prepareAction(
        win,
        displayText,
        current_connections,
        abstractArea );
    targetTime = System.currentTimeMillis()
      + Utilities.waitAndReturnSleeptime();
    found = false;
    while(System.currentTimeMillis()<targetTime && !found) {
      try {
        if ( current_connections.selenium.isElementPresent(
            "//span[@class='lesson_text_db' and" +
```

```
          "contains(.,'Jerry')]" ) ) {
        found = true;
      }
    } catch( Exception e ) {}
}
if (found) {
  successful = true;
} else {
  successful = false;
}
current_connections.utils.finishGeneratingAction(
   win,
   current_connections );
current_connections.utils.finishStep(
   win,
   abstractArea,
   waalArea,
   httpArea );
// end of the step

// begin of a new generation step
displayText = "viewProf(tom)" ;
// get connections for 'tom'
current_connections = getConnectionPool()
  .getConnectionsFor("jerry");
current_connections.utils.prepareAction(
    win,
    displayText,
    current_connections,
    abstractArea );
_commands = new LinkedList<WTCommand>();
_commands.add( new SelectItemsHTML(
    win,
    current_connections,
    "//select[@name='employee_id']",
    new String[]{"regexp:Tom.*"}) );
_commands.add( new ClickButtonHTML(
    win,
    current_connections,
    "//input[@value='ViewProfile']") );
while( true ) {
  try {
    current_connections.utils.executeWithOptions(
```

```
            win,
            current_connections,
            _commands );
      break;
    } catch (ExceptionNoMoreExamplesAvailable e) {
      // ask test expert
      // it has to provide manual http request information
      current_connections.utils
        .handleExeptionAskTestExpert(current_connections);
      break;
    } catch (ExceptionAskTestExpert e) {
      // ask test expert
      current_connections.utils
        .handleExeptionAskTestExpert(current_connections);
      break;
    }
  }
  current_connections.utils.finishGeneratingAction(
      win,
      current_connections );
  current_connections.utils.finishStep(
      win,
      abstractArea,
      waalArea,
      httpArea );
  // end of the step

  // begin of a new verification step
  displayText = "profile(tom)" ;
  // get connections for 'tom'
  current_connections = getConnectionPool()
    .getConnectionsFor("jerry");
  current_connections.utils.prepareAction(
      win,
      displayText,
      current_connections,
      abstractArea );
  targetTime = System.currentTimeMillis()
    + Utilities.waitAndReturnSleeptime();
  found = false;
  while(System.currentTimeMillis()<targetTime && !found) {
    try {
      if ( current_connections.selenium
```

```
                .isTextPresent( "792-14-6364" ) ) {
            found = true;
          }
        } catch( Exception e ) {}
      }
      if (found) {
        successful = true;
      } else {
        successful = false;
      }
      current_connections.utils.finishGeneratingAction(
          win,
          current_connections );
      current_connections.utils.finishStep(
          win,
          abstractArea,
          waalArea,
          httpArea );
      // end of the step
      // FINAL BLOCK: SHOW RESULT
      if( successful == true ) {
        System.out.println(
             "Verdict: Attack Trace successfully reproducible\n");
        Utilities.showMessage(
             "Verdict: Attack Trace successfully reproducible\n",
             !terminate);
      } else {
        System.out.println(
          "Verdict: Attack Trace NOT successfully reproducible\n");
        Utilities.showMessage(
          "Verdict: Attack Trace NOT successfully reproducible\n",
          !terminate);
      }
      if( terminate ) {
        System.out.println("Terminating TEE...");
        System.exit(0);
      }
    }
}
```