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 D4.1
SPaCIoS Tool mock up, Technology survey, Validation methodology patterns v.1

Abstract

In this deliverable, we provide a mock up of the SPaCIoS Tool, illustrating its main functionalities and how the user will interact with it, a survey of the existing technology, as well as a first version of a collection of general validation methodology patterns that comprises of two specific patterns corresponding to automated test case generation and interactive penetration testing.

Deliverable details

Deliverable version: *v1.1*
Classification: *public*
Date of delivery: *10.10.2011 (v1.0: 06.10.2011)*
Due on: *30.09.2011*
Editors: *all*
Total pages: *38*

Project details

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

SEVENTH FRAMEWORK PROGRAMME

(this page intentionally left blank)

Contents

List of Figures

List of Tables

1 Introduction

SPaCIoS has been developing and will develop a number of property-driven security testing and vulnerability-driven testing techniques. Automated support for these techniques will be provided by generating test cases with model checking and related automated reasoning techniques, applied to a model of the System Under Validation (SUV, although we will also synonymously speak of System Under Testing, SUT), the security goals, and a model of the attacker. The information that serves as input for the analysis (i.e., the security goals, the model of the attacker, and the model of the SUV) is typically provided by the Security Analyst, although in specific situations it will be possible to automatically derive the model of the SUV by taking advantage of behavioral service model specifications or by using model inference techniques that we are also developing.

In order to assess the effectiveness of the validation techniques developed within the project, we will implement and integrate them into the SPaCIoS Tool and apply the tool to a number of industrial-strength use cases from IoS scenarios. The architecture of the SPaCIoS Tool is depicted in Figure 1, and described in more detail in the following sections. In a nutshell, the tool takes as input a formal description of the SUV, the expected security goals, and a description of the capabilities of the attacker, and then automatically generates and executes a set of test cases on the SUV.

The architectural view of Figure 1 provides a preliminary and abstract description of the interplay among the techniques and libraries developed in the context of the project. In this deliverable, we present a more concrete architectural view of the tool that encompasses a description of the functionalities provided by the components as well as of their interfaces. In particular, in Section 2, we explain the main functionalities of the SPaCIoS Tool and their interplay with the Security Analyst and the SUV. This architecture will be used as a reference model for the integration that will be carried out in WP 4 (of which this deliverable is a part) of the prototype techniques and libraries developed in WP 2 and WP 3.

Then, in Section 3, we give a survey of existing penetration testing tools, along with the techniques used during the tests and the differences between tools. This survey results from a thorough analysis of the available open-source security testing tools that we carried out in order to assess the state-of-the-art technologies best suited to our task.

In Section 4, we provide a first version of a collection of general validation methodology patterns: we define two specific validation methodology patterns, corresponding to automated test case generation and interactive penetration testing.

We conclude by summarizing the main issues and giving a brief description of future work in WP 4 in Section 5, and by describing an attack methodology in Appendix A.

2 Overview of the SPaCIoS Tool

The SPaCIoS Tool mock up we present in this deliverable consists of a refined version of the architecture illustrated in the Description of Work, a presentation of the functional behavior of the component modules as well as of the workflow encompassing the internal functioning of the tool and its interaction with the Security Analyst and the SUV. The architecture of the SPaCIoS Tool is depicted in Figure 1, which refines the one given in the DoW and in previous deliverables. In Section 2.1, we present the components of the tool and then, in Section 2.2, we illustrate an example of interaction between the SPaCIoS Tool and the Security Analyst as well as that between the SPaCIoS Tool and the SUV. An example of application of the SPaCIoS Tool is given in [29, §10.2].

2.1 Components

User Interface. It is the interface component between the Security Analyst and the main functionalities offered by the SPaCIoS Tool. The Security Analyst uses the User Interface to (1) provide a formal model of the system under test and the environment, and the expected security properties, (2) define the correspondence between the formal model and the SUV, (3) retrieve information on the status of the SPaCIoS Tool, and (4) execute, monitor, and debug the tests.

Property-driven and Vulnerability-driven Testcase Generation. This module is in charge of the generation of the testcases starting from a formal model of the system under test and its environment and a description of the expected security property (or, dually, of a security weakness). The testcases generated are such that their execution should lead to the discovery of violation of the security property (or should confirm the existence of the security weakness, respectively).

Libraries There are four different categories of elements in the library component:

- *Vulnerabilities* (see [28])
- *Attack Patterns* (a set of rules for describing an attack [5])
- *Security goals* (see [28])
- *Attacker models* (see [28])

All these sets will be used as input for the *Property-driven and Vulnerability-driven test case generation* component. Moreover, Attack Patterns will be also used to guide the analyst in the iterative penetration testing phase (see Section 4.2) and in the refinement of abstract traces involving respectively the *User interface* and the *Test Execution Engine*. In both cases, Attack patterns will be used as inputs.

Model Inference and Adjustment. This component has the twofold task of building a formal model of the system under test and of the environment, and to adjust the available one. The construction of the model is necessary whenever no model is initially available. This is performed off-line, i.e. before starting the testcase generation and testing the SUV. Model adjustment is instead an online activity which is triggered when the execution of a test reveals a discrepancy between the model and the system under test and/or the environment.

Test Execution Engine. The testcases are finally executed by the Test Execution Engine (TEE) by handling the exchange of messages with the SUV. The TEE relies on the services offered by a Test Driver, an API offering a number of useful functionalities for parsing and generating the actual messages exchanged between SPaCIoS Tool and the SUV.

The internal architecture of the Property-driven and Vulnerability-driven Testcase Generation component is depicted in Figure 2. The functionalities offered of its sub-components is briefly described as follows.

Abstract Trace Generation via Model Checking. It takes a formal model of the SUV and of the environment (in the form of a labeled transition system) and the expected security properties (expressed as a LTL formula) as input and systematically explores the execution traces of the system looking for a violation of the properties. On demand, the model checker is also capable to enumerate all such execution traces.

Model Mutation. When the model-checker does not report any attack trace, potential faults are injected into the model such that the model-checker can find some attack traces to try on the system. The mutation operators represent known vulnerabilities that come from *a priori* knowledge.

Property Rewriting. When the model-checker does not report any attack trace for the LTL formula Φ that expresses the expected security property,

the property Φ is rewritten into an LTL formula Ψ. The property Ψ is in separated form, such that some future condition F implies some past condition P. Due to the specific choice of Ψ, each (finite) abstract trace in the model that can be extended to satisfy F has to satisfy P; else, the property Φ is violated. Therefore, by model checking $\neg F$ we find an abstract trace in the model that shows F is reachable (otherwise, the model satisfies Φ vacuously). By projecting the trace reaching F on the SUV, we check whether, or not, P holds in the trace exhibited by SUV.

Testcase Generation The abstract trace generated so far specifies the incoming and the outgoing messages for each step but it does not specify how they should be checked and generated, nor the way in which the internal state of the principals should be updated. For this reason the model is enriched with annotations that tell the TEE how to verify (respectively, generate) incoming (respectively, outgoing) messages by using the functionalities offered by the Test Driver. Moreover, additional steps can also be added in the abstract trace in order to test some specific vulnerabilities that can affect the SUV (vulnerability-driven testing). For instance, we envision that attack traces can be transformed in order to reproduce attack patterns available in the Libraries. Finally, the execution trace is projected on the input/output behavior of the SUV to produce the corresponding testcase.

2.2 Example of Interaction between the Security Analyst, the SPaCIoS Tool, and the SUV

An example of interaction between the Security Analyst, the SPaCIoS Tool, and the SUV is as follows:

1. *Formal Modeling.* The Security Analyst specifies a formal model of the system under test and of the environment along with the expected security properties. The analyst also specifies the scenario to be considered in the analysis, i.e. the set of involved principals and the roles played in the system.

2. *Model Inference or Adjustment.* If no model is provided by the Security Analyst, model inference techniques are applied in the attempt to automatically build a model of the system under test and of the environment; otherwise, execution continues with property-driven and vulnerability-driven testcase generation.

3. *Model Mutation.* Vulnerabilities are injected into the model.

4. *Property Rewriting.* The LTL formulae specifying the expected security properties as rewritten in separated form.

5. *Abstract Trace Generation.* Abstract traces are automatically generated by invoking a model checker. The model checker systematically explores the model looking for abstract traces witnessing the violation of the expected security properties. If no such a trace is found then execution terminates, otherwise these traces are returned. Notice that traces are abstract at this point, in the sense that only the pattern of the messages exchanged between the principals are specified.

6. *Definition of the SUV.* The Security Analyst specifies which principals are in the SUV and defines the correspondence between the model and the SUV. This amounts to specifying which entities of the model belong to SUV, the concrete configuration of SUV (i.e., the IP addresses and port numbers of services, etc.), as well as the information of the *Point of Control and Observation* (PCO) (i.e., which interface we can inject (observe) messages to (from, resp.)).

7. *Testcase Generation.* Abstract traces are refined into concrete testcases by adding program fragments that will be used during the execution of the test to check if incoming messages have the expected form and to generate the outgoing messages and by projecting the traces on the input/output behavior of the SUV.

8. *Test Execution.* The concrete execution traces are projected on and then executed against the SUT. If a discrepancy between the model and the actual system is detected, then the model inference and adjustment module is invoked to rectify the model.

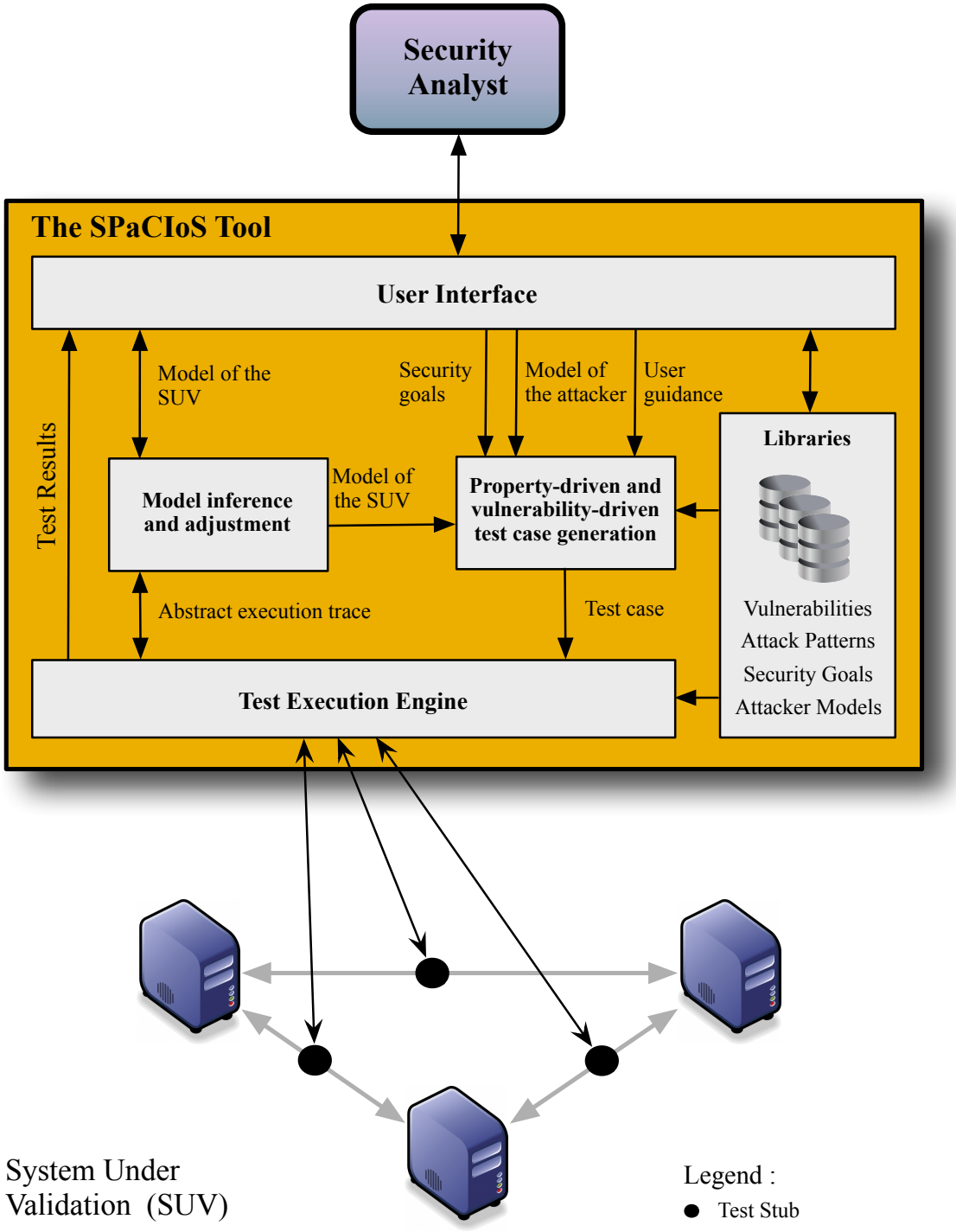

Figure 1: The SPaCIoS Tool and its interplay with the Security Analyst and the SUV

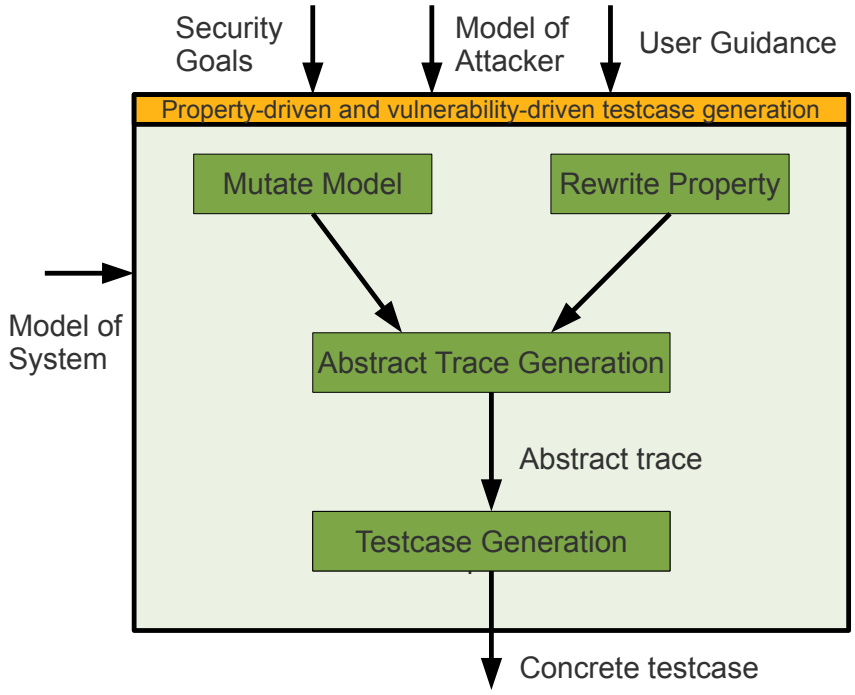

Figure 2: Architecture of the the Property-driven and Vulnerability-driven Testcase Generation Component

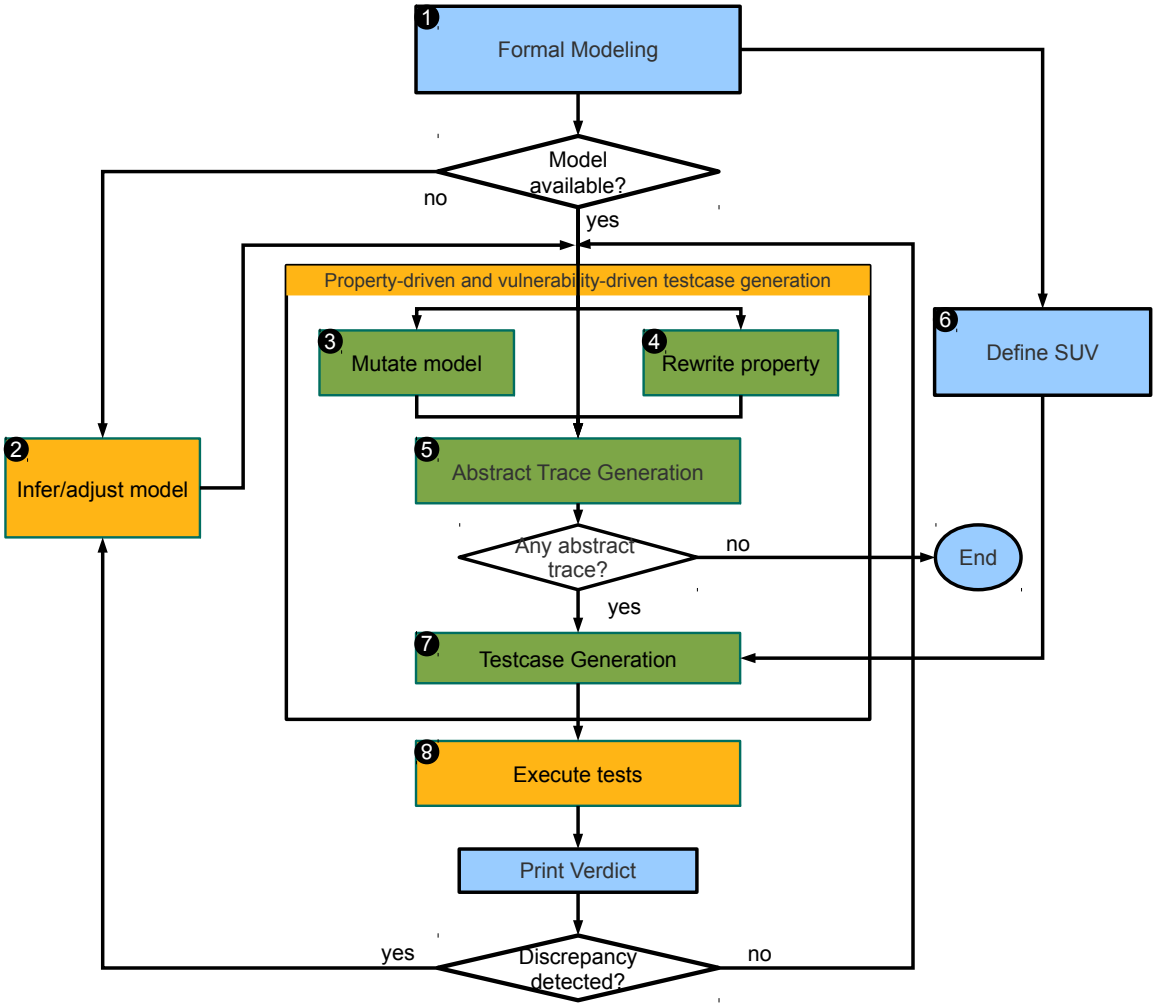

Figure 3: Interaction between the Security Analyst, the SPaCIoS Tool, and the SUV

3 Technology Survey

In this section, we give a survey about penetration testing tools, along with the techniques used during the tests and the differences between tools.

In Section 3.1, we present three main testing techniques used in several different tools. In Section 3.2, we give a first raw classification for security tools and we list some tools currently available to security analysts. We finally categorized these tools with respect to some parameters (like performing the real attack or API capability and so on) in Section 3.3.

3.1 Testing techniques

There exist several methodologies used by different tools for testing a system, for example: Black-box testing, source code analysis or model-based testing. In the following sections, we will give an overview of these approaches and we will categorize them based on the knowledge needed for using each tool (e.g. in black box testing we don't have the knowledge of the application source code).

3.1.1 Black-box testing (functional testing)

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

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

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

3.1.2 Source code analysis and white-box testing

Source code analysis (see, e.g., [6]) is the process of checking source code for coding problems based on a fixed set of patterns or rules that might indicate possible security vulnerabilities. This process is a subset of white-box testing (see, e.g., [15]), which is a method of testing applications by checking the structure starting from the source code.

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

of our project. Their investigation will be carried out in the final two years of SPaCIoS, especially thanks to the addition of the new partner IeAT (as described in the amended description of work effective from October 1, 2011).

3.1.3 Model-based testing

Model-Based Testing (MBT) consists in a variant of software testing in which a model of the system under testing (SUT) is used to derive test cases, namely pairs of inputs and (expected) outputs, for its implementation. The purpose of MBT, as for software testing, is to generate a test suite (a set of test cases) accordingly to a specific criterion and to execute it on the SUT in order to acquire more confidence about the correct behavior of a system implementation, or to discover failures (i.e., unexpected behaviors). Advantages of adopting MBT in contrast of other testing approaches that do not rely on an abstract model of the SUT, are many-fold and mainly related to the involvement of Model Checkers in the testing process. The most important advantage is the possibility to generate test cases having a specific purpose in an automated way, thanks to the capability of the Model Checker to provide attack traces. It is indeed possible to formalize the purpose of a test suite in terms of goals and use them, with the model of the SUT, in order to cast the test case generation problem as a model checking problem. For example, in the context of coverage testing, one can generate abstract tests by using goals checking the execution of transitions. By doing so, the Model Checker will provide every execution trace including such transition.

In the context of SPaCIoS, we are interested in security testing and we are going to use two techniques of MBT, namely Mutation and LTL Separation (which are described in Section 4).

In general, MBT covers three majors tasks: automatic generation of abstract test cases from models, concretization of abstract tests in order to obtain executable tests, and theirs execution on the SUT by using manual or automatic means.

3.2 Tools type

In the previous section we have seen which methods are used by tools in order to test applications. In this section we will use a different approach. In fact, as parameters, we will consider what kind of information is retrieved during test and what kind of tests are performed.

Following the partition from [34] we have categorized tools as following:

- port scanners,

- vulnerability scanners,
- application scanners,
- web application assessment proxy,
- packet sniffer.

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

3.2.1 Port Scanners

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

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

3.2.2 Vulnerability Scanners

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

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

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

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

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

3.2.3 Application Scanners

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

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

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

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

3.2.4 Web Application Assessment Proxy

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

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

3.2.5 Packet sniffer (protocol analyzer)

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

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

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

3.3 Tools survey

Following the initial categorization given in Section 3.1 and Section 3.2, we have collected some tools in Table 1.

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

After this previous (generic) distinction we will go a little bit into details. In Table 2 we have categorized tools listed in Table 1 with respect to the ability of performing some kind of actions.

From the data collected in Table 2, together with data from Table 1, some interesting facts arise:

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

3.4 Conclusion

In this section we have given an overview of the most used security tools, regarding their use and their capabilities. From this point of view, the SPaCIoS Toolwill have characteristic in common with many tools we have previously

	A	B	C	D	E
Nmap [19]	X				
Scapy [4]	X				
Tenable Nessus [22]		X			
Core Impact [33]		X			
Qualys's QualysGuard [25]		X			
ISS's Internet Scanner [25]		X			
Nikto [31]		X			
Wikto [18]		X			
Maltego [20]		X			
SPI Dynamics's WebInspect [10]			X		
Rational Appscan [13]			X		
N-STEALTH [21]			X		
Metasploit [26]			X		
Canvas [14]			X		
Acunetix free ed wvs [1]			X		
Hailstorm [12]			X		
Beef [3]			X		
Wapiti [32]			X		
owasp lapse [16]			X		
Paros Proxy [7]				X	
OWASP's WebScarab [23]				X	
Burpsuite [17]				X	
Acunetix wvs [2]				X	
Grendel-Scan [11]				X	
PAROS pro desktop [7]				X	
Selenium [27]				X	
Ettercap [8]					X
Firesheep [9]					X
Wireshark [35]					X

Table 1: Some tools categorized into: A - Port scanners, B - Vulnerability scanners, C - Application scanners, D - Web application assessment proxy, E - Packet sniffer.

appointed. We can say that the SPaCIoS Toolwill be a sort of hybrid, taking the best from all the tools nowadays available, and adding feature like the testcase generation via model checking or interactive penetration testing.

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

Table 2: Some tools categorized into: A - Scripting / API capabilities, B - handle a model, C - perform attacks.

4 Validation methodology patterns

As the architecture of the SPaCIoS tool suggests (see Section 2), there are different approaches and techniques employed together in order to achieve the project objectives. All these techniques constitute the *Validation methodology patterns* of the SPaCIoS project and consequently of the tool we will implement in the next two years.

The validation methodology encompasses the *automated test case generation* and the *interactive penetration testing* respectively described in Section 4.1 and in Section 4.2. The first methodology comprises three techniques for the generation of test cases, Model-Checking based Testing, Mutation and LTL separation. The second methodology is based on an attack methodology which is adopted in order to provide guidelines to perform a penetration testing on the SUV.

4.1 Automated test case generation

Model-Checking based Testing

As a security validation pattern, the following steps are involved in the model-checking based testing:

- Formal Modeling. The first step is the definition of an ASLan++ specification of SUV with expected security properties. This is done in a way that (1) the size of the model could be handled by model-checker, (2) the system aspects relevant for the expected security properties are present in the model, and (3) the attack traces returned by the model checker (if any) contain information useful to automate the execution of the test. The Refinement Mapping between the formal model and the actual system is also specified.

- Model Checking. The ASLan++ specification is submitted to the model checker, which may find an Abstract Attack Trace as the result of the violation of certain security properties. In case the expected security properties are not violated, this security validation pattern terminates.

- Trace Improvement. More transitions are added into Abstract Attack Trace to obtain a more specific attack trace, the Improved Attack Trace. This is done by taking into account (1) the guidance of the security analyst, (2) the Refinement Mapping, namely a set of information characterizing the SUT, (3) the formal model, and (4) the libraries of the SPaCIoS tools.

- Model Instrumentation. The model is annotated with information about how to (1) generate messages for the SUV, (2) parse messages from the SUV, and (3) update the internal state of the principals. 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 Test Execution Engine in order to (1) calculate the concrete value of a (for messages delivered to SUV), or (2) checking the compliance of a concrete message with the structure of the expected one (for messages coming from SUV). ℓ_a is a pointer to an internal location of the SPaCIoS tool where the corresponding value, obtained by executing p_a, is stored.

- Test Execution. The Test Execution Engine takes Refinement Mapping, Instrumented Model, and Improved Attack Trace as input, executes the test by simulating the principals of the model that do not belong to the SUV and by executing the attack trace on the SUV. As the result, a test verdict is obtained.

Mutations

When the model of the SUV is *secure*, the model-checker does not report any attack trace. In order to test the SUT, we need to introduce potential faults in the model such that the model-checker finds some attack traces to try on the system. The fault injection is done via mutation operators at the model level. Instead of using any mutation operator allowed by the description language (ASLan++), a set of mutation operators is selected that represent known vulnerabilities coming from *a priori* knowledge.

The current mutation operators we have studied so far all reflect authorization flaws. They are based on manipulating checks at the model level that should guarantee authorization properties. In practice this could be reflected by accepting any value for a received variable instead of a specific value, or by commenting out security-related conditions based on fact statements. Unfortunately the current mutation operators do not apply to any ASLan++ model in general. Therefore we are looking for ways to generalize this but the most promising direction is to provide guidelines to modelers. Thus, such mutation operators could be automatically applied to any ASLan++ model that follows those guidelines.

LTL Separation

If the model of the SUV satisfies the desired security properties, then model-checking is unable to produce any Abstract Attack Trace. This does not necessarily mean the SUV satisfies the desired properties, as there may be a

discrepancy between the model and the SUV. LTL separation can then be used to generate abstract test cases by checking the model of the SUV with respect to "separated" security properties.

A desired security property is expressed in the form of an LTL formula Φ. LTL separation is used to obtain a formula Ψ in a separated form, such that some future condition F implies some past condition P. Due to the specific choice of Ψ, each (finite) Abstract Trace in the model that can be extended to satisfy F has to satisfy P; else, the property Φ is violated.

This information reveals which Abstract Traces are more likely to have concretizations in the SUV violating the security property. Moreover, for each such Abstract Trace, there is a corresponding finitely testable condition given by P.

4.2 Interactive penetration testing

In addition to the techniques described above, we also envision the possibility to stress the SUV in order to perform a penetration testing following an attack methodology. We are willing to adopt the attack methodology described in [30], integrated with the library of attack patterns, as one of the methodologies to validate the SUV with the SPaCIoS tool. This is possible thanks to the fact that an attack strategy can be reused as a way to test if the SUT implements countermeasures capable of protecting its assets from common types of attack. In other words, if all different attempts to attack the SUV fails, the security analysts become more confident about the correctness of the SUV with respect to the security requirements it is supposed to meet.

The *interactive penetration testing* validation pattern is depicted in Figure 4 where the big arrow indicates the order of steps composing the attack strategy e.g. *analyze the application* has to be executed before *test the client-side controls*. Other arrows indicate that elements of the *library of attack patterns* can be used in order to narrow subtasks composing the steps of the methodology: sequences of actions belonging to an attack pattern can be adopted to complete subtasks and, as explained in the following, to guide the analysts in performing penetration testing. Indeed, we envision the possibility to guide security analysts in the penetration testing process step-by-step thanks to the adoption of attack patterns, given that a current state is provided to the SPaCIoS tool from the analyst. Namely, given the set of steps already performed (from the attack methodology) and the interactions with the SUV already occurred, it could be possible to look into the set of libraries of attack patterns in order to provide security analysts with a list of suggested steps. These suggested steps will be a possible way to

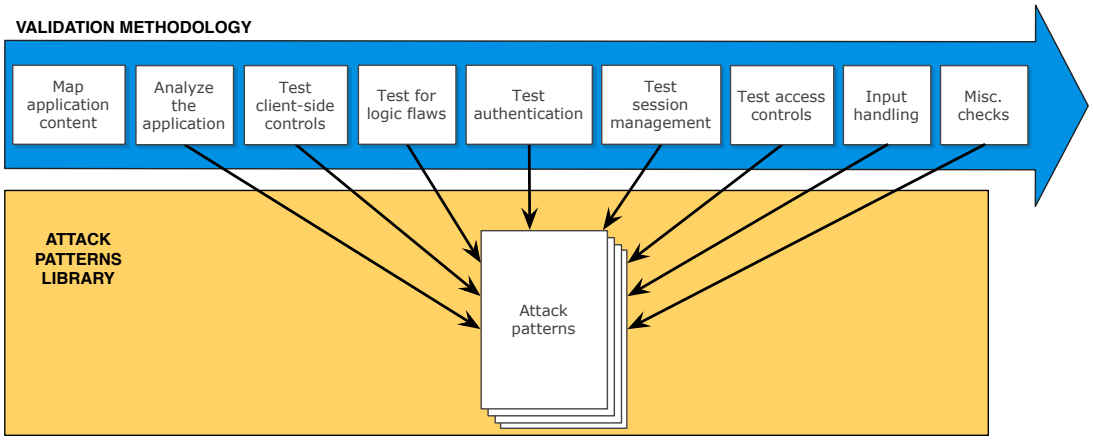

Figure 4: Validation methodology: penetration testing approach using the library of attack patterns

proceed with the penetration testing, accordingly to the attack methodology described in Appendix A.

As already mentioned, the attack methodology described in Appendix A reflects the approach followed in [30] to attack a Web application. Although the techniques we considered could be seen as a subset of the ones contained in [24], they are nevertheless a core set suitable for the current stage of the SPaCIoS project. In particular, they will be the basis for defining the first set of attack patterns in the library component of the SPaCIoS tool (see Figure 1). We are also going to extend the described techniques in order to cover as much as possible the penetration testing features of [24] in the remaining two years of the project.

5 Conclusion

We have presented an overview of the SPaCIoS Tool with a short description of each component, as well as its interaction with the Security Analyst and the SUV. In addition to clarifying the role of the different components of the architecture of the tool, this deliverable will provide a reference model for the integration that will be carried out in WP 4 of the prototype techniques and libraries developed in WP 2 and WP 3.

We have also given a survey about penetration testing tools, along with the techniques used during the tests and the differences between tools. We expect that this will provide the basis for a more in depth comparison of the SPaCIoS Tool with such tools, as well as possible cross-fertilization of ideas and combination/integration of SPaCIoS with other tools.

The first version of the collection of general validation methodology patterns that we have given (comprising of automated test case generation and interactive penetration testing) will be extended in future deliverables with additional patterns together with a specification of the use case scenarios for which they are best suited. We expect that other, more sophisticated patterns, lying in the continuum between automated test case generation and interactive penetration testing, will be identified as the SPaCIoS Tool will be applied against the problem cases identified in WP 5.

A Attack methodology

Identify application content The first step of attacking an application is to gather information about it, in order to gain a better understanding of what one is up against.

1. Discover content:

   - Hidden content: Confirm how the application handles requests for "nonexistent items". Obtain listings of common file and directory names, and common file extensions. Try to understand the naming conventions used by application developers. Review all client-side code to identify any clues about hidden server-side content, including HTML comments and disabled form elements.

   - Default content: Detect any default or well-known content that is present.

2. Non-standard access methods

   - Identifier-specified functions: Identify any instances where specific application functions are accessed by passing an identifier of the function in a request parameter. Compile a list of common function names or cycle through the syntactic range of identifiers observed to be in use.

   - Debug parameters: Choose one or more application pages or functions where hidden debug parameters may be implemented. Review the application's responses for any anomalies that may indicate that the added parameter has had an effect on the application's processing.

Analyze the application

1. Identify the core functionality that the application was created for.

   - Identify the core security mechanisms employed by the application and the ways they work.

   - Understand the key mechanisms that handle authentication, session management, and access control, and the functions that support them, such as user registration and account recovery.

   - Identify all of the more peripheral functions and behavior (e.g. redirections, off-site links, error messages, and administrative and logging functions).

2. Identify data entry points

   - Identify all of the different entry points that exist for introducing user input into the applications processing, including URLs, query string parameters, POST data, cookies, and other HTTP headers processed by the application.
   - Examine any customized data transmission or encoding mechanisms used by the application.
   - Identify any out-of-band channels via which user-controllable or other third-party data is being introduced into the application's processing.

3. Identify technologies

   - Identify each of the different technologies used on the client side.
   - As far as possible, establish which technologies are being used on the server side.
   - Try to fingerprint the web server.
   - Identify any interesting-looking script names and query string parameters that may belong to third-party code components.

4. Identify the attack surface.

   - Try to ascertain the likely internal structure and functionality of the server-side application and the mechanisms that it uses behind the scenes to deliver the behavior that is visible from the client perspective.
   - For each item of functionality, identify the kinds of common vulnerabilities that are often associated with it.
   - Formulate a plan of attack, prioritizing the most interesting looking functionality and the most serious of the potential vulnerabilities associated with it.

Test client-side controls

1. Transmission of data via client

- Hidden fields: Locate all instances within the application where hidden form fields, cookies, and URL parameters are apparently being used to transmit data via the client. Modify the item's value in ways that are relevant to its role in the application's functionality. Determine whether arbitrary values submitted in the field are processed by the application, and whether this can be exploited to interfere with its logic or subvert any security controls.

2. Test Client-Side Controls over User Input

- JavaScript validation or length limit: Identify any cases where client-side controls such as length limits and JavaScript checks are used to validate user input before it is submitted to the server. Test each affected input field in turn by submitting input that would ordinarily be blocked by the client-side controls, to verify whether these are replicated on the server.

Test for logic flaws

- Multi-stage processes. When a multistage process involves a defined sequence of requests, attempt to submit these requests out of the expected sequence.

- Incomplete input. For critical security functions within the application, which involve processing several items of user input and making a decision based on these, test the application's resilience to requests containing incomplete input.

- Trust boundaries. Probe the way the application handles transitions between different types of trust of the user. Look for functionality where a user with a given trust status can accumulate an amount of state relating to their identity. Try to find ways of making improper transitions across trust boundaries by accumulating relevant state in one area and then switching to a different area in a way that would not normally occur.

- Transaction logic. In cases where the application imposes transaction limits, test the effects of submitting negative values.

Test authentication

1. Direct attacks

   - Test password quality. For example we can check for very short or blank password, common dictionary words or names, password set to the same as the username or still set to a default.
   - Test impersonation functions: some application implements the facility for a privileged user of the application to impersonate other users, in order to access data and carry out actions within their user context.
   - Test username uniqueness. If the application has a self-registration function that lets you specify a desired username, attempt to register the same username twice with different passwords.
   - Check for unsafe transmission of credentials. Walk through all authentication-related functions that involve transmission of credentials, including the main login, account registration, password change, and any page that allows viewing or updating of user profile information. Monitor all traffic passing in both directions between the client and server.

2. Authentication logic

   - Test for Fail-Open Conditions: For each function in which the application checks a user's credentials, including the login and password change functions, walk through the process in the normal way, using an account you control. Note every request parameter submitted to the application. Modifying each parameter in turn in various unexpected ways designed to interfere with the application's logic.
   - Test Any Multistage Mechanisms: If any authentication-related function involves submitting credentials in a series of different requests, identify the apparent purpose of each distinct stage, and note the parameters submitted at each stage. Repeat the process numerous times, modifying the sequence of requests in ways designed to interfere with the application's logic. Determine whether any single piece of information (such as the username) is submitted at more than one stage, either because it is captured more than once from the user or because it is transmitted via the client in a hidden form field, cookie, or preset query string parameter. If so, try submitting different values at different stages (both valid and invalid), and observing the effect. Try to exploit the application's behavior to gain unauthorized access or reduce the effectiveness of the controls imposed by the mechanism.

Test session management

1. Token generation

   - Test for meaning: some session tokens are created using a transformation of the user's name or email address, or the information associated with them.
   - Test for predictability. The type of potential variations one might encounter here are open ended, but the authors indicates that predictable session token commonly arise from three different sources: i) concealed sequences, ii) time dependency, and iii) weak random number generation.

2. Token handling

   - Check for insecure transmission
   - Check for disclosure in logs. If your application mapping exercises identified any logging, monitoring, or diagnostic functionality, review these functions closely to determine whether any session tokens are disclosed within them.
   - Test mapping of tokens to sessions.
     - Log in to the application twice using the same user account, either from different browser processes or from different computers. Determine whether both sessions remain active concurrently. If so, the application supports concurrent sessions, enabling an attacker who has compromised another user's credentials to make use of these without risk of detection.
     - Log in and log out several times using the same user account, either from different browser processes or from different computers. Determine whether a new session token is issued each time, or whether the same token is issued each time the same account logs in. If the latter occurs, then the application is not really employing proper session tokens at all, but is using unique persistent strings to re-identify each user. In this situation, there is no way to protect against concurrent logins or properly enforce session timeout.
     - If tokens appear to contain any structure and meaning, attempt to separate out components that may identify the user from those

that appear to be inscrutable. Try to modify any user-related components of the token so that they refer to other known users of the application, and verify whether the resulting token i) is accepted by the application, and ii) enables you to masquerade as that user.

- Test session termination. After logging out, attempt to reuse the old token and determine whether it is still valid by requesting a protected page using the token. If the session is still active, then users remain vulnerable to some session hijacking attacks even after they have "logged out".
- Test for session fixation. If the application issues session tokens to unauthenticated users, obtain a token and perform a login. If the application does not issue a fresh token following a successful login, then it is vulnerable to session fixation.
- Check for XSRF (Cross-site request forgery). If the application relies solely upon HTTP cookies as its method for transmitting session tokens, then it may well be vulnerable to cross-site request forgery attacks.

Test access controls

- Understand the requirements. Based on the core functionality implemented within the application,understand the broad requirements for access control, in terms of vertical segregation (different levels of user having access to different types of functionality) and horizontal segregation (users at the same privilege level having access to different subsets of data).
- Testing with multiple accounts.
  - If the application enforces vertical privilege segregation, first use a powerful account to locate all of the functionality that it can access, and then use a less-privileged account and attempt to access each item of this functionality.
  - If the application enforces horizontal privilege segregation, perform the equivalent test using two different accounts at the same privilege level, attempting to use one account to access data belonging to the other account. This typically involves replacing an identifier (such as a document ID) within a request to specify a resource belonging to the other user.

  - When you perform any kind of access control test, be sure to test every step of multistage functions individually, to confirm whether access controls have been properly implemented at each stage, or whether the application assumes that users who access a later stage must have passed security checks implemented at the earlier stages.

- Test for insecure methods.

Input handling

1. Code injection The topic of code injection is a huge one, encompassing dozens of different languages and environment, and a wide variety of different attacks.

- SQL injection (see http://ha.ckers.org/sqlinjection/).
- XSS & response injection (see http://ha.ckers.org/xss.html).
  - Reflected XSS.
  - Stored XSS.
  - DOM-based XSS.
- OS command injection.
- Path traversal: many kind of functionality oblige a web application to read from or write to a file system on the basis of parameters supplied within user requests. If these operations are carried out in an unsafe manner, an attacker can submit crafted input which causes the application to access files that the application designer did not intend it to access.
- Script injection.
- File inclusion.

2. Test for function-specific input vulnerabilities

- SMTP injection.
- Native code flaws.
- SOAP injection.
- LDAP injection.
- XPath injection.

Miscellaneous checks

1. Test for shared hosting issues

   - Test segregation in shared infrastructure
   - Test segregation between ASP-hosted applications

2. Test the web server

   - Test for default credentials. For any identified interfaces, consult the manufacturer's documentation and common default password listings to obtain default credentials. If you gain access to an administrative interface, review the available functionality and determine whether this can be used to further com promise the host and attack the main application.
   - Test for dangerous HTTP methods. Use the OPTIONS method to list the HTTP methods that the server states are available. Note that different methods may be enabled in different directories.
   - Test for proxy functionality. Using both GET and CONNECT requests, try to use the web server as a proxy to connect to other servers on the Internet, and retrieve content from them.
   - Test for virtual hosting misconfiguration.
   - Test for web server software bugs

3. Test for DOM-based attacks. Perform a brief code review of every piece of JavaScript received from the application to identify any XSS or redirection vulnerabilities that can be triggered by using a crafted URL to introduce malicious data into the DOM of the relevant page. Include all standalone JavaScript files and scripts contained within HTML pages (both static and dynamically generated). Identify all uses of the following APIs, which may be used to access DOM data that is controllable via a crafted URL. Trace the relevant data through the code to identify what actions are performed with it. If the data (or a manipulated form of it) is passed to one of the following APIs, then the application may be vulnerable to XSS. If the data is passed to one of the following APIs, then the application may be vulnerable to a redirection attack.

4. Test for frame injection. If the application uses frames, review the HTML source of the main browser window, which should contain the code for the frameset. Look for <frame> tags which contain a name attribute. If any are found, then the application is potentially vulnerable to frame injection.

5. Follow up information leakage. In all of your probing of the target application, monitor its responses for error messages that may contain useful information about the cause of the error, the technologies in use, and the application's internal structure and functionality.

6. Test for weak SSL ciphers. If any weak or obsolete ciphers and protocols are supported, then a suitably positioned attacker may be able to perform an attack to downgrade or decipher the SSL communications of an application user, gaining access to their sensitive data.

References

[1] Acunetix. Acunetix web vulnerability scanner. http://www.acunetix.com/.

[2] Acunetix web application security. http://www.acunetix.com/vulnerability-scanner/.

[3] Beef: The browser exploitation framework project. http://beefproject.com/.

[4] Philippe Biondi. Scapy. http://www.secdev.org/projects/scapy/.

[5] CAPEC. *CAPEC – Common Attack Pattern Enumeration and Classification, release 1.6*. The MITRE Corporation, 2010. Available at http://capec.mitre.org/.

[6] Steven R. Lavenhar Christoph Michael and Howard F. Lipson. Source Code Analysis Tools - Overview. https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/263-BSI.html, 2009.

[7] Chinotec Technologies Company. Paros - for web application security assessment. http://www.parosproxy.org/.

[8] Ettercap. http://ettercap.sourceforge.net/.

[9] Firesheep. http://codebutler.github.com/firesheep/.

[10] HP Fortify. Hp webinspect. https://www.fortify.com/products/web_inspect.html.

[11] Grendel-scan. http://grendel-scan.com/.

[12] Cenzic hailstorm professional. http://www.cenzic.com/products/cenzic-hailstormPro/.

[13] IBM. Rational appscan. http://www-01.ibm.com/software/awdtools/appscan/.

[14] Immunity Inc. Immunity canvas. http://www.immunitysec.com/products-canvas.shtml.

[15] Girish Janardhanudu and Ken van Wyk. White Box Testing. https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box/259-BSI.html, 2009.

[16] Lapse: The security scanner for java ee applications. https://www.owasp.org/index.php/OWASP_LAPSE_Project.

[17] PortSwigger Ltd. Burp suite. http://portswigger.net/burp/.

[18] SensePost Pty Ltd. Wikto. http://www.sensepost.com/labs/tools/pentest/wikto.

[19] Gordon Lyon. Nmap security scanner. http://www.nmap.org/, 2011.

[20] Maltego. http://www.paterva.com/web5/.

[21] N-Stalker. N-stalker web application security scanner. http://www.nstalker.com/products/editions/.

[22] Tenable network security. Tenalbe nessus. http://www.nessus.org/products/nessus.

[23] OWASP. *OWASP WebGoat and WebScarab*. OWASP, 2007. Available at http://www.lulu.com/product/file-download/owasp-webgoat-and-webscarab/1889626.

[24] OWASP. OWASP_Testing_Guide_v3. https://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf, 2008.

[25] Qualysguard it security. http://www.qualys.com/products/qg_suite/.

[26] Rapid7. Metasploit framework. http://www.metasploit.com/.

[27] Selenium. http://seleniumhq.org/.

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

[29] SPaCIoS. Deliverable 5.1: Proof of Concept and Tool Assessment v.1, 2011.

[30] Dafydd Stuttard and Marcus Pinto. *The web application hacker's handbook: discovering and exploiting security flaws*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[31] Chris Sullo and David Lodge. Nikto. http://www.cirt.net/nikto2.

[32] Nicolas Surribas. Wapiti. http://wapiti.sourceforge.net/, 2006.

[33] Core Security Technologies. Core impact. http://www.coresecurity.com/content/core-impact-overview.

[34] Kenneth R. van Wyk. Penetration Testing Tools. https://buildsecurityin.us-cert.gov/bsi/articles/tools/penetration/657-BSI.html, 2007.

[35] Wireshark. http://www.wireshark.org/.