Practical experience gained from passive testing of Web based systems.

Alessandra Bagnato, Fabio Raiteri
Corporate Research Divisions
TXT e-solutions
16100 Genoa, Italy
{alessandra.bagnato, fabio.raiteri}@txt.it

Wissam Mallouli, Bachar Wehbi
Montimage EURL
75013 Paris, France
{wissam.mallouli, bachar.wehbi}@montimage.com

Abstract — **In recent years Web-based systems have become extremely popular and, nowadays, they are used in critical environments such as financial, medical, and military systems. As the use of Web applications for security-critical services has increased, the number and sophistication of attacks against these applications have grown as well. For this reason it is essential to be able to prove that the target Web-based system implements its designed security requirements avoiding known vulnerabilities in HTTP-based solutions. To reach this aim, we can rely on several testing techniques and mainly on security passive testing approach that is becoming increasingly important to security-relevant aspects into web based software systems. This article describes the application of the TestInv-P passive testing tool as part of the testing phase of TXT e-tourism Web application. TestInv-P is a passive testing tool that monitors communication traces of an application during run-time and verifies whether it satisfies certain security-related invariants derived from SHIELDS models.[1]**

Keywords-testing: passive testing; Web based application; security requirements; invariants; practical experience.

I. Introduction

In modern networks, the heterogeneity and the increasing distribution of applications, such as telecommunication protocols, Web-based systems and real-time systems, make security management complex. These applications are more and more open and rely on networking parts of computer systems that generally make use of different security solutions. Secure Software engineering is assuming an increasingly importance to guarantee and to correctly apply quality assurance techniques for preventing security defects across the entire software development lifecycle (SDLC) [2].

The testing phase is one of the crucial phases in the SDLC that allows checking whether software respects its security requirements and is out of any known vulnerability. Tests are then carried out to check if some known vulnerabilities would remain present. Even if several tools for some specific tests (such as passwords crackers) exist, there is no general solution analyzing the overall system conformance according to its security requirements. Several reasons can explain these deficiencies. First, there is currently few research work on formal modeling of complete security policies, even if some aspects, such as access control security rules, have been studied further. In addition, analytical work about security checking often focuses on the verification of punctual elements, such as cryptographic protocols or code analysis. Thus, the responsible for security and all the system administrators are missing a formal solution to ensure the coherence of a system implementation with respect to its security requirement, even if this last has been fairly well defined.

Once security requirements allowing known vulnerabilities avoidance (called also mitigations) are formally specified, it is essential to prove that the target system implements these requirements. Indeed, if one cannot ensure this conformance, the global security cannot be guaranteed anymore. Many solutions can be proposed to achieve this objective (the implementation conformance with respect to its security requirements). The conformance guarantee can be reached for instance by:

- formally injecting the security policy in the considered system code,
- or by formally specifying the target system to prove that it verifies the security requirements it has to respect,
- or by considering several strategies of formal tests.

This last methodology will be explored in this paper and mainly the passive testing technique ([6],[3]) that aims at observing, during the run-time, whether the system behaviour conforms to its security specification formally described as invariants and derived from SHIELDS security models. This conformance passive testing procedure follows these four steps:

Step 1: Properties formulation. The relevant protocol properties to be tested are provided by the standards or by protocol experts.

Step 2: Properties as invariants [3]. Properties have to be formulated by means of invariants that express local properties; i.e. related to a local entity. Moreover, the properties are formally verified on the formal specification

[1] 'SHIELDS' is an FP7 project concerned with model-based detection and elimination of software vulnerabilities. In this project, we conduct research and development on models for software vulnerabilities and security countermeasures, develop a repository where such models can be stored, and extend and adapt security and development tools to make use of this repository.

ensuring that they are correct with respect to the requirements. These invariants can be downloaded directly from the SHIELDS SVRS[2] [8] to be used by a non-initiated tester. These invariants may need to be adapted before being able to be used in testing some specific applications and services.

Step 3: Extraction of execution traces. In order to obtain such traces, different OP (Observation Points) are set up by means of a network sniffer installed on one of the system serves. The captured traces are in XML format.

Step 4: Test of the invariants on the traces. The traces are processed in order to obtain information concerning particular events as well as relevant data (e.g. source and destination address, origin of data to initialize a variable, etc.). During this processing, the test of the expected properties is performed and a verdict is emitted (Pass, Fail or Inconclusive). An inconclusive verdict may be obtained if the trace does not contain enough information to allow a Pass or Fail verdict. Notice that processing the collected communications can be also done on-the-fly by applying online passive testing.

Thus, best practices or principles of secure software engineering can be modeled as an invariant. A tester (human) with little security background (e.g., a software developer) is able to perform passive testing with TestInv-P if the invariant are given (more details about TestInv-P is provided in section II).

This article discusses experience collected during the use of TestInv-P approach. We discuss the experiences gained from the process of testing in the final development stage of the e-tourism project by TXT e-solutions S. p. A.

The following research questions were addressed by applying passive testing technology on a TXT case study:

Question 1: To which extent are non-experts able to use the tool?

Question 2: Is the approach sufficient to identify security flaws during testing?

Question 3: How much effort has to be put into creating new invariants?

The rest of the paper is organized as follows. The section II present the background on the passive testing approach using TestInv-P and a general overview of modeling invariants as part of the development lifecycle in software engineering. Section II introduces the basics about passive testing and the TestInv-P tool. Section III introduces the TXT scenario: the e-tourism project. Section IV introduces invariants development. Section V and Section VI describe the modeling of invariants and the usage of TestInv-P in practice. Lessons learned are addressed in Section VII. Section VIII summarizes our work and describes future work.

[2] The SHIELDS SVRS is a centralized repository that allows the storage and sharing security models in order to reduce known security vulnerabilities during software development.

II. Passive Testing Basics and Tool

A. *Passive versus Active Testing Methodologies*

Conformance testing for security checking usually relies on the comparison between the behavior of an implementation and its formal security specification by checking whether they are equivalent. Two main approaches are commonly applied: passive and active testing.

Active testing [7],[11] is based on the execution of specific test sequences against the implementation under test. These test sequences are commonly generated from the formal security and functional models. They may be generated automatically or semi-automatically from formal models that represent test criteria, hypothesis and test goals. These sequences (with an executable format) are performed by establishing Points of Control and Observation (PCO, execution interfaces) defined by the testers. Nevertheless, when testing a black-box implementation, only few PCOs are available and the results are therefore compromised.

In the other hand, passive testing [1],[4],[6] consists in observing the exchange of messages (input and output events) of an implementation under test during run-time. The term passive means that the tests do not disturb the natural operation of the protocol. The record of the observed events is called a trace. This trace will be compared to security properties derived from the standard or proposed by the protocol experts. The passive testing techniques are applied, in particular, because they are non-intrusive whereas the active testing techniques need to stimulate the system under test and may cause its crush. In addition, passive testing can be applied on a system in its real context (with real users) whereas active testing is usually performed in a simulated environment with simulated/emulated system users.

In this work, we present and apply a passive testing approach to check the security of Web based systems. This invariant-based approach is illustrated using TestInv-P passive testing tool and its usage in the particular case of the "e-tourism" Web application.

B. *Passive testing tool*

TestInv-P tool [10] aims at passively testing a deployed communicating system under test (SUT) to verify if it respects a set of properties called invariants [3]. In the case of TestInv-P, invariants describe the correct order of exchanged messages among system entities with conditions on communicated data.

Originally TestInv-P tool was conceived to check whether the SUT collected communication traces respect a set of functional invariants without considering the security requirements. The expressivity of invariants [10] has been improved during the SHIELDS project to focus on security aspects and also so that they can be derived from more abstract SHIELDS models. For improving security of the application TestInv-P has been applied in the testing phase of the SDLC.

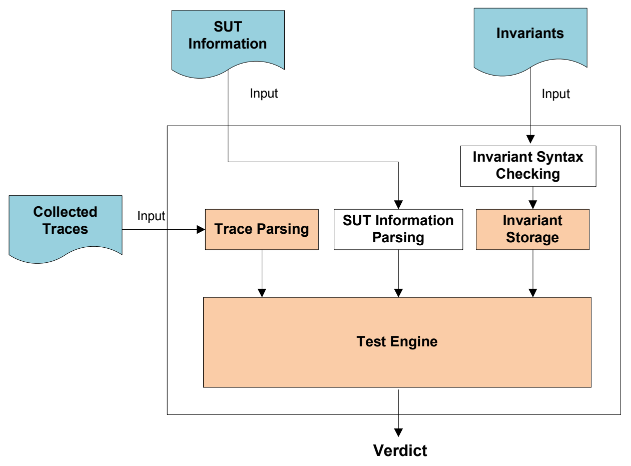


Figure 1. TestInv-P architecture for security checking

Figure 1 illustrates the components of the TestInv-P tool. It has three different inputs:

- Information on the system under test that is being observed. This information represents data of interest (protocol packets fields' names for example) that are relevant to the automated analysis of the captured traces.
- The SUT invariants defined in XML format. By invariants we mean here the combination of conditions that must be respected by the system. The non-respect of an invariant may imply vulnerability.
- And the communication traces represented in XML format (captured using Wireshark for instance).

In order to use the TestInv-P, the first step consists of defining the invariants and the data of interest. This can be done by an expert of the system under test that understands in details the studied protocol or service. The invariants are then checked with respect to their XML schema. The next step consists of capturing the communication traces using Wireshark and analysing them using the TestInv-P tool.

The TestInv-P tool, developed by Montimage[3], allows automated analysis of the captured traces to determine if the given invariants are satisfied or not. The tool is written in C and uses the XML2 library to interpret the traces and the invariants.

C. Passive testing algorithm

In this section we present the algorithm that allows the deduction of a verdict by analyzing the system trace with respect to a set of the predefined **obligation invariants.** The obtained verdict for an invariant can be either: PASSED, FAILED or INCONCLUSIVE meaning respectively that all events[4] where satisfied, that at least one event was not satisfied or that it is not possible to give a verdict due to the lack of information in the trace.

[3] Montimage is French SME located in Paris working in the testing field. More in www.montimage.com

[4] An event within an invariant is a set of conditions on relevant fields of captured packets in the trace.

The algorithm is used by the tool and analyses the traces in, at most, time complexity of $O(N^2)$ or to be more precise: the number of packets that need to be analyzed is N^2 x I x T where N = the number of packets in the trace, I = the number of invariants and T = the average time spent in analyzing an invariant on a packet. This can be reduced to N x I x T if we store information of each condition for each packet in a hash table making it necessary to inspect each packet only once. This will be done for the future version of the tool that will thus be able to perform correctly for on-line invariant analysis.

ALGORITHM: TestInv-P invariant analysis

START OF ALGORITHM

INPUT: Invariants (I_1,...,I_k), trace (P_1,...,P_N). Each invariant I_i is a set of events denoted ($e_{\{i,R\}}$,$e_{\{i,1\}}$, ... $e_{\{i,n(i)\}}$) where $e_{\{i,R\}}$ is the triggering event, ($e_{\{i,1\}}$, ..., $e_{\{i,n(i)\}}$) are the events that need to be satisfied before $e_{\{i,R\}}$ and n(i) is the number of these events.

OUTPUT: Verdicts (V_1,...,V_k)

```
(1) FOR EACH packet $P_t$ in trace (from t=N to t=1) DO
(2)     /* the algorithm parses the trace from the end */
(3)     r = $P_t$
(4)     /* r is the reference packet */
(5)     SET all current event of invariants $E_1$, ..., $E_k$ to first event $e_1$
(6)     FOR EACH invariant $I_i$ (from i=1 to i=k) DO
(7)         $V_i$ = SKIP
(8)         IF r satisfies $e_{\{i,R\}}$
(9)     /* r satisfies an event e if all the conditions of e are satisfied
(10)    by the packet r */
(11)            $V_i$ = INCONCLUSIVE
(12)            current $I_i$ = $e_{\{i,1\}}$
(13)    /* current $I_i$ indicates the event that needs to be inspected
(14)    at a given moment (this means that, for this invariant,
(15)    the events before have been satisfied,
(16)    the others not yet) */
(17)        ENDIF
(18)    ENDFOR
(19)    IF no INCONCLUSIVE found
(20)        continue
(21)    ENDIF
(22)    FOR EACH $P_{\{t-s\}}$ (from s=1 to s=t-1) DO
(23)        c = $P_{\{t-s\}}$
(24)        /* c is the current packet */
(25)        FOR EACH invariant $I_i$ (from i=1 to i=k) DO
(26)            IF ($V_i$ = PASSED or $V_i$ = FAILED
                   or $V_i$ = SKIP)
(27)                continue
(28)            ENDIF
(29)            IF c satisfies current $I_i$
(30)                IF current $I_i$ = $e_{\{i,n(i)\}}$
(31)                    $V_i$ = PASSED
(32)                    continue
(33)                ELSE
(34)                    current $I_i$ =
next.current $I_i$
(35)        /* current $I_i$ has been satisfied so the next time
(36)        around we need to inspect the next
(37)        event for this invariant */
(38)                ENDIF
(39)            ELSIF (timestamp(c) - timestamp(r)) >
maxTime
```

```
(40)                    V_i = FAILED
(41)                    continue
(42)                ENDIF
(43)            ENDFOR
(44)        ENDFOR
(45)        FOR each invariant I_i (from i=1 to i=k) DO
(46)            IF V_i ≠ SKIP
(47)                print V_i and information about r
(48)            ENDIF
(49)        ENDFOR
(50) ENDFOR

END OF ALGORITHM
```

The algorithm operation is the following: starting at the end of the trace, for each packet (that we call **reference** packet) we inspect the **<if>** clause of each invariant (from line 2 to 18). If any passes then we inspect all the previous packets (that we call **current** packets) to determine if the **<then>** clause is satisfied (from line 22 to 44). The **<then>** clause is composed of a series of reversed ordered events than need to have happened sometime (limited by the timeout) before the **reference** packet. Thus we treat all the packets as **reference** packets and determine which invariants pass or not for each.

The algorithm that allows checking simple invariants on a captured trace is a variant of this proposed algorithm. The only difference is located in the sense of parsing of the trace. In the case of simple invariants, we begin the parsing from the beginning of the file and move forward until the end of the trace.

III. The TXT Scenario

The following section describes the context of the TestInv-P usage in the SDLC testing phase of a web based project at TXT. TXT aimed at developing a web accessible distributed repository for storing digitalized cultural data provided by users. The system to be developed has to provide a complete mobile e-tourism information system for travelers and is intended to encourage exploration [12].

Within the e-tourism system, users are able to share, through a web interface, their impressions and experiences and to exchange content with other users via chat, forums, blogs, image and video sharing, etc. That information is intended to be used by other people when planning their itinerary. The system has then to suggest location-based information and an ad-hoc itinerary planning functionality. Such information has to be collected and presented via web technologies. Furthermore, a user management system has to be established to prevent misuse.

In addition, users should obtain web based reliable and secure information on geographic coordinates (i.e., georeferential data) concerning all nearby locations with points of interest.

Information managed by the system comprises personal information and sensitive data. Consequently, security models had already been introduced to cover security issues in the same scenario in the requirement analysis phase of the SDLC [5].

IV. Invariant Development

This section describes the principles behind the construction of Invariants. For creating invariants, an appropriate level of security expertise is indispensible across the team. Therefore, at least the invariant developer in the team should have sufficient knowledge and practical experience in security.

The invariants are written in XML format to make it easier to interpret by humans and software. They are represented in an IF-THEN structure (see Figure). Basically, if in the trace we find the events that satisfy the triggering context, then we need to verify that the events of the Verdict Clause are part of the trace as well.

```
...

<invariant name="INVARIANT 1">
<if>
  <event reference="TRUE">
    <condition>
      <variable>Type</variable>
      <operation>=</operation>
      <value>REPLY</value>
    </condition>

    ...
  </event>
</if>
<then type="BEFORE" max_skip="-1" max_time="10">
  <event verdict="TRUE">
    <condition>
      <variable>Type</variable>
      <operation>=</operation>
      <variable>REQ</value>
    </condition>
    <condition>
      <variable>Source</variable>
      <operation>=</operation>
      <variable
type=="REFERENCE">Destination</value>
    </condition>

    ...
  </event>
</then>
</invariant>

...
```

Figure 2. XML format for defining an invariant

Within an invariant, the <if> tag identifies the triggering events. An event is a set of conditions that need to be satisfied for a given packet.

When analyzing an invariant we have the triggering packet, called “REFERENCE” packet, and the other packets, called “CURRENT” packets, that should occur “BEFORE” (in the case of an obligation invariant) or “AFTER” (in the case of a simple invariant) the triggering packet. The events that need to be verified on the "CURRENT" packets are found in the <then> tag.

Conditions over the packets are conditions on data of interest of these packets. They are of the form:

(<variable>)(<operation>)(<variable>|<value>)

Values can be strings or numbers and operations can be: contains, is equal to, does not contain, is different than, is less than, is less or equal than, etc. Variables can refer to the "REFERENCE" packet or the "CURRENT" packet. This allows comparing the two variable values (for instance we can check if the "REFERENCE" packet has the same call id as the packet that is currently being analyzed).

The definition of invariants also includes some attributes needed by the "Test engine" module to deduce a verdict. For an invariant to PASS the distance in time ("timeout") and/or in number of packets ("skipout") between the "REFERENCE" packet (that triggered the invariant) and the "CURRENT" packet (that satisfied the last event) is respectively limited by max_time and max_skip attributes values. This is necessary for two reasons:

1) the tool stores packet information and defining limits allows reducing the memory needed, and

2) if no limit is given then all invariants that are not satisfied for a given trace are "INCONCLUSIVE" because the packets that are needed could have occurred before the beginning of the trace or after.

It must be noted here that some "INCONCLUSIVE" invariants could occur if the triggering "REFERENCE" packet is close to the beginning or the end of the trace if the trace does not start from the initial state or end in the final state. The invariants are stored in a C structure that constitutes an input for the "Test engine" module.

V. Modeling Invariants in practice

The system must ensure that user credentials (i.e., data for authentication purposes) are stored securely and that all relevant data are not accessible to unauthorized parties. This requirement is very important in the analyzed context, since information entered by users is to be shared and trusted by the whole community, and it is mandatory to test that the information provided on the site originates from a person with essential knowledge of the subject.

Starting from security requirements of the e-tourism project, a set of invariants can be deduced (12 invariants have been used). In the following, we present a selection of invariants used in the context of e-tourism Web system.

- **The invariant 1** expresses that a secure POST request (expecially in login submission) should not contain a clear text password, this is because a malicious attacker (man in the middle) may catch this clear password and use it to get access on the web site with that stolen identity.
- **The invariant 2 and 3** express that a HTTPS protocol should be used in order to exchange sensitive packets between client and server in a more secure way; invariant 2 is applied to the TCP destination port, while invariant 3 is applied to the TCP source port.
- **The invariant 4** expresses that if a system received a response message (indicating the availability of a given service), this means that it has already sent a request message to discover the same service. Indeed, a malicious system may send a large number of response messages to over consume the resources on the peer system. This can lead to a Deny of Service (DoS) attack. The invariants are described in the "invariant.xml" file.

Other invariants dealing with other DoS attacks are used but will not be presented in detail in this paper. These invariants describe the five basic types of DoS attacks [13]:

- Consumption of computational resources, such as bandwidth, disk space, or processor time
- Disruption of configuration information, such as routing information.
- Disruption of state information, such as unsolicited resetting of TCP sessions.
- Disruption of physical network components.
- Obstructing the communication media between the intended users and the victim so that they can no longer communicate adequately.

In the following, we provide more details about the 4 invariants presented above as well as their XML format. Invariant 1 searches within the input trace file (the pdml file with packets information) for POST requests packets that contain clear text password as follows:

```
…
  <invariant name="INVARIANT 1: POST request
    should not contain clear text password">
     <if>
        <event reference="TRUE" verdict="TRUE">
           <condition>
              <variable>http.request.method</variable>
              <operation>=</operation>
              <value>POST</value>
           </condition>
        </event>
     </if>
     <then type="BEFORE" max_skip="-1"
       max_time="10">
        <event verdict="TRUE">
           <condition>
              <variable type="REFERENCE">
                 data-text-lines
              </variable>
              <operation>not_contains</operation>
              <value>pass</value>
           </condition>
        </event>
     </then>
  </invariant>
…
```

Invariant 2 searches within the input trace file for packets usage of the HTTPS protocol (on destination port).

```
…
  <invariant name="INVARIANT 2: HTTPS should be
      used">
      <if>
          <event reference="TRUE" verdict="TRUE">
              <condition>
                  <variable>tcp.dstport</variable>
                  <operation>contains</operation>
                  <value>http</value>
              </condition>
          </event>
      </if>
      <then type="SAME" max_skip="-1"
         max_time="10">
          <event verdict="TRUE">
              <condition>
                  <variable type="REFERENCE">
                      tcp.dstport
                  </variable>
                  <operation>not_contains</operation>
                  <value>80</value>
              </condition>
          </event>
      </then>
    </invariant>
…
```

Invariant 3 search within the trace file for packets usage of the HTTPS protocol (on source port).

```
…
  <invariant name="INVARIANT 3: HTTPS should be
      used">
      <if>
          <event reference="TRUE" verdict="TRUE">
              <condition>
                  <variable>tcp.srcport</variable>
                  <operation>contains</operation>
                  <value>http</value>
              </condition>
          </event>
      </if>
      <then type="SAME" max_skip="-1"
         max_time="10">
          <event verdict="TRUE">
              <condition>
                  <variable type="REFERENCE">
                       tcp.srcport
                  </variable>
                  <operation>not_contains</operation>
                  <value>80</value>
              </condition>
          </event>
      </then>
    </invariant>
…
```

Invariant 4 searches within the input trace file for Denial of Service (DoS) attempts and if they are well managed.

```
…
<invariant name="INVARIANT 4: Denial Of Service
      (DOS) should be managed">
      <if>
          <event reference="TRUE" verdict="TRUE">
              <condition>
                  <variable>http.response.code</variable>
                  <operation>=</operation>
                  <value>200</value>
              </condition>
          </event>
      </if>
      <then type="BEFORE" max_skip="-1"
        max_time="100">
          <event verdict="TRUE">
              <condition>
                  <variable>http.request.method</variable>
                  <operation>=</operation>
                  <value>M-SEARCH</value>
              </condition>
              <condition>
                  <variable>ip.src</variable>
                  <operation>=</operation>
                  <variable type="REFERENCE">
                       ip.dst
                  </variable>
              </condition>
              <condition>
                  <variable>http.st</variable>
                  <operation>=</operation>
                  <variable type="REFERENCE">
                      http.st
                  </variable>
              </condition>
          </event>
      </then>
    </invariant>
…
```

VI. USING TESTINV-P IN PRACTICE

In this section, TestInv-P is applied to the e-tourism project developed at TXT. This mechanism permits to automatically detect available remote systems and their provided services. The e-tourism project uses the HTTP protocol since it consists of an advanced web site.

The e-turism developers have followed four main phases :

- The first one was dedicated to understanding how to use the tool and what was needed in order to get start with the test; this phase took us a couple of hours.
- The second phase was dedicated to the creation of the HTTP traces (pdml files through wireshark) that took us half an hour.
- The third phase was dedicated to the execution of TestInv-P on the traces.
- The final phase wad dedicated to the interpretation of the results generated by TestInv-P in which we opened all resulting files seeking for HTTP problems or weaknesses.

The testing has been focused on five different critical actions that within the website navigation could be affected by errors due to a bad programming or by missing security requirements.

These 5 critical actions are:

1) Registration: This is the usual process performed by users to get registration to the web site.

2) Login: This is basically the action in which the user inserts his username and password in order start the http session and log in as registered user .

3) Itinerary Creation: This action consists of a creation of a tourism itinerary on the embedded map.

4) Password Request: This is the usual system used to get a forgotten password through an email automatic process.

5) Logout: This is essentially the opposite of the action described above, that is the action in which the user session is closed.

Using Wireshark, a network packet sniffing tool, a communication trace has been captured and stored in different files. In the following, a selection from the "Login.pdml" is provided.

Service discovery pdm trace in XML format (a selection only):

```
<?xml version="1.0"?>
<pdml version="0" creator="wireshark/1.0.8">
  <packet>
    <proto name="geninfo" pos="0" showname="General
        information" size="78">
      <field name="num" pos="0" show="1"
        showname="Number" value="1" size="78"/>
      <field name="len" pos="0" show="78"
      <field name="timestamp" pos="0" show="May 29,
      2009 16:09:33.673767000" showname="Captured
      Time" value="1243606173.673767000" size="78"/>
    </proto>
    <proto name="frame" showname="Frame 1 (78 bytes
on
    wire, 78 bytes captured)" size="78" pos="0">
      <field name="frame.time" showname="Arrival Time:
       May 29, 2009 16:09:33.673767000" size="0"
pos="0"
       show="May 29, 2009 16:09:33.673767000"/>
      <field name="frame.time_delta" showname="Time
       delta from previous captured frame: 0.000000000
       seconds" size="0" pos="0" show="0.000000000"/>
      …
    </proto>
  </packet>
  <packet>
    …
  </packet>
  …
</pdml>
```

In the E-Tourism case study we analyzed a total of 6266 packets for the 9 operation (Audio Tour, Event Calendar Navigation, Forum & Blog, Itinerary Creation, Login, Logout, Map Navigation, Password Request, Registration); however we took into account only 5 of them for this work, the 5 we hold as most important and relevant (Itinerary Creation, Login, Logout, Password Request, Registration; corres-ponding to a total of 998 packets).

Number of packets for operation:

Audio Tour = 164;
Event Calendar Navigation = 876;
Forum & Blog = 1128;
Map Navigation = 3100; (high number of packet since it is longer the time spent on that section)
Itinerary Creation = 528;
Login = 79;
Logout = 74;
Password Request = 144;
Registration = 173;

Data of interest are defined by an expert of system under test and are stored in "traceformat.properties" file as follows:

```
Data of interest for service discovery mechanism:

timestamp : show
ip.src : show
ip.dst : show
tct.srcport : showname
tcp.dstport : showname
http.request.method: show
http.response.code : show
data-text-lines : show
```

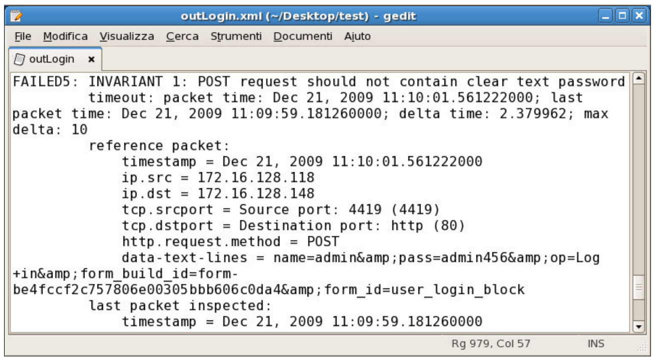
In order to check whether the invariants are respected by the captured trace of the e-tourism project, we have run very easily the TestInv-P tool using the following command:

```
[user@localhost test]# ./passive_test
-t <full path name>/Login.pdml
-i <full path name>/invariants.xml
-k <full path name>/traceformat.properties
–s <full path name>/"outLogin.xml"
```

In the command, <full path name> should be replaced with full path name of directory where the corresponding file is located and invariants.xml should contain the properly developed invariants.

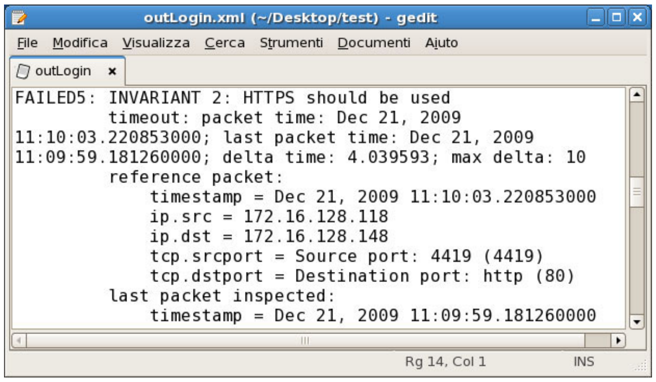
Figure 3, 4, 5 and 6 show selections of the resulting output file (outLogin.xml) given from execution of TestInv-P on Login.pdml trace file; it is possible to see that invariant 1, invariant 2, invariant 3 and invariant 4 have been failed on some relevant packages.

A verdict is then generated for each invariant. The result contains information that identifies the network packets involved in the invariants violation as well as the type of violation:


outLogin.xml (~/Desktop/test) - gedit
File Modifica Visualizza Cerca Strumenti Documenti Aiuto
outLogin
Rg 979, Col 57
INS

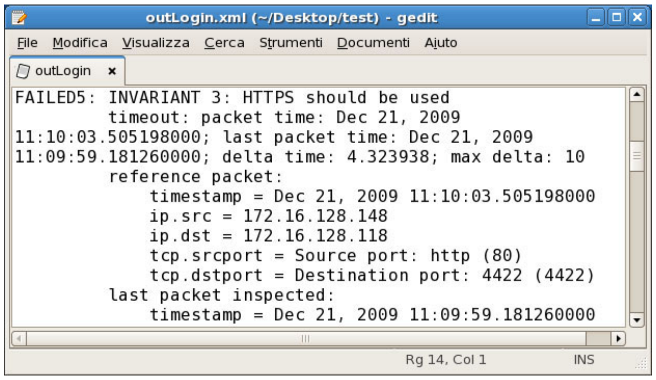
```
FAILED5: INVARIANT 1: POST request should not contain clear text password
         timeout: packet time: Dec 21, 2009 11:10:01.561222000; last
packet time: Dec 21, 2009 11:09:59.181260000; delta time: 2.379962; max
delta: 10
         reference packet:
             timestamp = Dec 21, 2009 11:10:01.561222000
             ip.src = 172.16.128.118
             ip.dst = 172.16.128.148
             tcp.srcport = Source port: 4419 (4419)
             tcp.dstport = Destination port: http (80)
             http.request.method = POST
             data-text-lines = name=admin&amp;pass=admin456&amp;op=Log
+in&amp;form_build_id=form-
be4fccf2c757806e00305bbb606c0da4&amp;form_id=user_login_block
         last packet inspected:
             timestamp = Dec 21, 2009 11:09:59.181260000
```

Figure 3. TestInv-P output screenshot for invariant 1


outLogin.xml (~/Desktop/test) - gedit
File Modifica Visualizza Cerca Strumenti Documenti Aiuto
outLogin
Rg 14, Col 1
INS

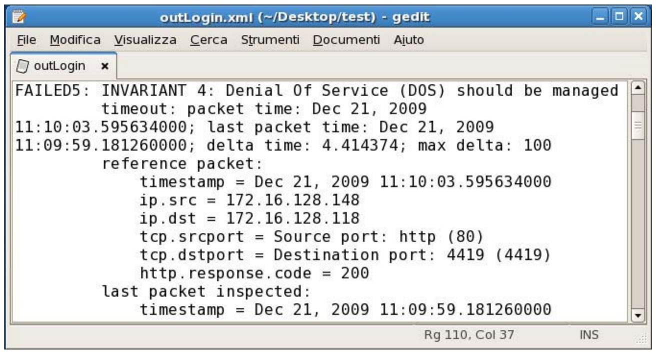
```
FAILED5: INVARIANT 2: HTTPS should be used
         timeout: packet time: Dec 21, 2009
11:10:03.220853000; last packet time: Dec 21, 2009
11:09:59.181260000; delta time: 4.039593; max delta: 10
         reference packet:
             timestamp = Dec 21, 2009 11:10:03.220853000
             ip.src = 172.16.128.118
             ip.dst = 172.16.128.148
             tcp.srcport = Source port: 4419 (4419)
             tcp.dstport = Destination port: http (80)
         last packet inspected:
             timestamp = Dec 21, 2009 11:09:59.181260000
```

Figure 4. TestInv-P output screenshot for invariant 2


outLogin.xml (~/Desktop/test) - gedit
File Modifica Visualizza Cerca Strumenti Documenti Aiuto
outLogin
Rg 14, Col 1
INS

```
FAILED5: INVARIANT 3: HTTPS should be used
         timeout: packet time: Dec 21, 2009
11:10:03.505198000; last packet time: Dec 21, 2009
11:09:59.181260000; delta time: 4.323938; max delta: 10
         reference packet:
             timestamp = Dec 21, 2009 11:10:03.505198000
             ip.src = 172.16.128.148
             ip.dst = 172.16.128.118
             tcp.srcport = Source port: http (80)
             tcp.dstport = Destination port: 4422 (4422)
         last packet inspected:
             timestamp = Dec 21, 2009 11:09:59.181260000
```

Figure 5. TestInv-P output screenshot for invariant 3


outLogin.xml (~/Desktop/test) - gedit
File Modifica Visualizza Cerca Strumenti Documenti Aiuto
outLogin
Rg 110, Col 37
INS

```
FAILED5: INVARIANT 4: Denial Of Service (DOS) should be managed
         timeout: packet time: Dec 21, 2009
11:10:03.595634000; last packet time: Dec 21, 2009
11:09:59.181260000; delta time: 4.414374; max delta: 100
         reference packet:
             timestamp = Dec 21, 2009 11:10:03.595634000
             ip.src = 172.16.128.148
             ip.dst = 172.16.128.118
             tcp.srcport = Source port: http (80)
             tcp.dstport = Destination port: 4419 (4419)
             http.response.code = 200
         last packet inspected:
             timestamp = Dec 21, 2009 11:09:59.181260000
```

Figure 6. TestInv-P output screenshot for invariant 4

TestInvP found several FAULT invariant instances on the packets, as reported below

- In Login:
  - o Use HTTPS instead of HTTP (invariant 2, invariant 3) – was not included in requirements
  - o POST request should not contain clear text password (invariant 1) – was not included in requirements
  - o Denial Of Service (DOS) should be managed (invariant 4) – needed adjustment due to programming error
- In Logout:
  - o Use HTTPS instead of HTTP (invariant 2, invariant 3) – was not included in requirements
- In Itinerary Creation:
  - o Use HTTPS instead of HTTP (invariant 2, invariant 3) – was not included in requirements
  - o POST request should not contain clear text password (invariant 1) – was not included in requirements
  - o Denial Of Service (DOS) should be managed (invariant 4) – needed adjustement due to programming error
- In Password Request:
  - o Use HTTPS instead of HTTP (invariant 2, invariant 3) – was not included in requirements
  - o POST request should not contain clear text password (invariant 1) – was not included in requirements
- In Registration:
  - o Use HTTPS instead of HTTP (invariant 2, invariant 3) – was not included in requirements
  - o POST request should not contain clear text password (invariant 1) – was not included in requirements

These results are then followed up by developers of e-tourism to improve the reliability of the product and avoid potential vulnerabilities that can be exploited up malicious users and attackers. Much of the issues found with TestInv-P results where caused by flaws within the requirements, https protocol was not considered within the first implementation of the e-turism. The first version was planned to run just with http. While some others, like the Denial Of Service (DOS) have been found to have to be managed solving a bug within the implementation as specified by the invariants creators team (DoS level 1 caused by bombarding a server with requests).

VII. Lessons Learned

Three developers from TXT industry partner without specific security knowledge were involved in the TestInv-P usage creation process and three developers of invariants were involved in Montimage, one of them with much security expertise, two with security-specific expertise.

Question 1, "to which extent are non-experts able to use the tool" has been positively answered; the TXT developers were able to use the tool and to identify security issues without having been involved into the creation of the invariants. Question 2 "Is the approach sufficient to identify security flaws during testing" has also been positively answered, the approach was considered very helpful to detect security flaws during the testing phase of the software development life cycle in particular in relation to the network protocol aspects to which the TestInv-P tool was intended. The TXT e-tourism project is currently running a second phase of testing taking into account the identified security flaws.

The main lessons learned from the experiment is that TestInv-P tool seems to be easy to learn/use even by non-initiated developer, but invariants are difficult to select, adapt and use (should be done by a security and formalism expert). This last point answered to the research question number 3, "How much effort has to be put into creating new invariants", we found that security expertise is absolutely mandatory for invariants creation. But this task can be easier if we refer to other security SHIELDS models (like the mitigations of misuse cases [8]). Moreover the results of the tool seem to be quite difficult to analyze and some known vulnerability elimination strategies need to be added to facilitate the developer task. In addition, a user interface for the tool will obviously help developers to better exploit the tools results (the presentation of the results will be improved). Nevertheless, the tool makes users more productive in finding security flaws and avoiding then potential attacks.

VIII. Conclusion and Future Work

In this article, we have summarized the experiences gained by introducing the passive testing tool TestInv-P in an industrial environment. The results of the tool are promising and allow detecting security vulnerabilities based on formal invariants. Using TestInv-P can improve the customer trust as more secure software with improved quality can be produced. In addition, security aspects are tested in a faster way using the TestInv-P tool with detailed focus on specific protocol aspects.

It has been considered as very important the fact that also developers not expert in security were able to easily use the tool in the testing phase of the software development lifecycle. At this stage the difficulties related to the selection, usage and adaption of invariants remain an issue to be treated by security expert but that was seen as an advantage for the acceptance of the tools by the developers' involved first testing phase of the developed code.

Currently, the TestInv-P tool can be applied to traces captured off-line. In the near future it will also be able to analyse the protocol exchange on the fly for applications and protocols that produce a reasonable rate of packets per second. A user interface will also be developed to help users to perform the test and understand their corresponding results.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement n° 215995 (SHIELDS).

References

[1] César Andrés, Mercedes G. Merayo, Manuel Núñez: Passive Testing of Timed Systems. ATVA 2008.

[2] C. Banerjee and S. K. Pandey, Software Security Rules, SDLC Perspective, In CoRR journal, volume abs/0911.0494, 2009.

[3] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on in-variants: application to the WAP. In Computer Networks, volume 48, pages 247-266. Elsevier Science, 2005.

[4] Ana Rosa Cavalli, Azzedine Benameur, Wissam Mallouli, Keqin Li, A Passive Testing Approach for Security Checking and its Practical Usage for Web Services Monitoring ,9ème Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE'09), Montréal, Canada, June 29 - July 3, 2009.

[5] Christian Jung, Frank Elberzhager, Alessandra Bagnato, Fabio Raiteri Practical Experience gained from Modeling Security Goals. Using SGITs in an Industrial Project. To be published In: Proceedings of the 4th International Workshop on Secure Software Engineering (ARES-SecSE 2010), Andrzej Frycz Modrzewski Cracow College Krakow, Poland, February 15th - 18th 2010.

[6] David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu, Xia Yin: Network protocol system monitoring: a formal approach with passive testing. IEEE/ACM Trans. Netw. (TON) 14(2):424-437, 2006.

[7] Keqin Li, Laurent Mounier, Roland Groz: Test Generation from Security Policies Specified in Or-BAC. COMPSAC 2007.

[8] SHIELDS D3.1: Initial repository specification and design, deliverable of the SHIELDS research project within the European Community's Seventh Framework Programme, 2008, http://www.shields-project.eu

[9] SHIELDS Project Consortium. D2.1 Formalism definitions and representation sche-mata. SHIELDS Project Deliverable D2.1. http://www.shields-project.eu/

[10] SHIELDS Project Consortium. D4.3 Final report on inspection methods and prototype vulnerability recognition tools. SHIELDS Project Deliverable D4.3. http://www.shields-project.eu/

[11] W. Mallouli, M. Lallali, G. Morales and A.R. Cavalli: Modeling and Testing Secure Web-Based Systems: Application to an Industrial Case Study, The fourth International Conference on Signal-Image technology & Internet-Based Systems (SITIS 2008), Bali, Indonesia, November 30 - December 03, 2008.

[12] M. Megliola, L. Barbieri, "Integrating Agent and Wireless Technologies for location-based Services in Cultural Heritage", Digital Cultural Heritage - Essential for Tourism, 2nd EVA Conference, 2008

[13] Tao Peng, Christopher Leckie, Kotagiri Ramamohanarao: Survey of network-based defense mechanisms countering the DoS and DDoS problems. ACM Comput. Survey (CSUR) 39(1), 2007.