A Systematic Approach to Web Application Penetration Testing Using TTCN-3

Bernard Stepien, Pulei Xiong, and Liam Peyton

School of Information Technology and Engineering,
University of Ottawa, Canada
{bernard,xiong,lpeyton}@site.uottawa.ca

Abstract. Penetration testing is critical for ensuring web application security. It is often implemented using traditional 3GL web test frameworks (e.g. HttpUnit, HtmlUnit). There is little awareness in the literature that a test specification language like TTCN-3 can be effectively combined with such frameworks. In this paper, we identify the essential aspects of TTCN-3 for penetration testing and how best to use them. These include separating abstract test logic from concrete data extraction logic, as well as support for templates, matching test oracles and parallel test components. The advantages of leveraging TTCN-3 together with 3GL web test frameworks for penetration testing is demonstrated and evaluated using example scenarios. The work was performed with a prototype TTCN-3 tool that extends the TTCN-3 model architecture to support the required integration with 3GL web test frameworks. A concrete proposal for modifying the TTCN-3 standard to support this refinement is described.

Keywords: web application security, model-based testing, penetration testing, test specification, TTCN-3.

1 Introduction

Web application vulnerabilities have been exploited since the early '90s against user oriented applications such as email, online shopping, and Web banking. Testing for web application vulnerabilities continues to be a significant problem, as more and more user-oriented applications are deployed to the web such as Facebook and Twitter Blog. It is often implemented using traditional 3GL web test frameworks (e.g. HttpUnit [11], HtmlUnit [10], JUnit [12]). There is little awareness in the literature that a test specification language like TTCN-3 [5] can be effectively combined with such frameworks. In this paper, we identify the essential aspects of TTCN-3 for penetration testing and how best to use them. These include separating abstract test logic from concrete data extraction logic, as well as support for templates, matching test oracles and parallel test components.

The use of a test specification language like TTCN-3 can improve the quality of the test oracles or assertions. General purpose language (GPL) approaches to penetration testing tend to be problematic because test oracles have to be pre-defined, and verification is limited to spot checking on a limited number of web page elements. As a result of this, confidence is reduced on the completeness of the test results which

G. Babin, K. Stanoevska-Slabeva, P. Kropf (Eds.): MCETECH 2011, LNBIP 78, pp. 1–16, 2011.
© Springer-Verlag Berlin Heidelberg 2011

means there is always a lingering concern that some vulnerability has gone undetected. As well, there is poor test automation re-usability both for a given application (regression testing) and between various applications (conceptual generalization).

The advantages of leveraging TTCN-3 together with 3GL web test frameworks for penetration testing is demonstrated and evaluated in this paper in section 2 (separating test logic from data extraction logic) and section 3 (test oracles) using two concrete attacks against the web application vulnerabilities: SQL Injection [17] (an attack occurs on server-side) and Persistent Cross Site Scripting [21](an attack occurs on client-side).

The work was performed with a prototype TTCN-3 tool that extends the TTCN-3 model architecture to support the required integration with 3GL web test frameworks. A concrete proposal for modifying the TTCN-3 standard to support this refinement is described in section 4. An existing TTCN-3 vendor has already incorporated the changes in the latest version of their tool.

2 Background and Related Work

A vulnerability is a bug or misconfiguration that can be exploited [14]. Penetration testing detects vulnerabilities in a system by attempting to recreate what a real attacker would do [22]. Penetration testing is often implemented using a general purpose language combined with test frameworks such as Metasploit [15], AttackAPI [3], as well as special browser extensions e.g. Firebug [8] and GreaseMonkey [9]. Usually a general purpose programming language is used with the frameworks and specialized tools to automate test execution.

A description of current approaches to penetration testing can be found in [16, 1, 18]. There are many factors that affect test case coverage and quality of testing. In [2] it was found that the tester's knowledge, skills and experience are factors. The resources available to testers are also relevant [4]. Other research has proposed test methodology changes to ensure testing is conducted more systematically and efficiently by, for example, integrating penetration testing into a security-oriented development life cycle [19]. Our use of test specifications written in TTCN-3 fits well with this approach. [13] provides an example of passive intrusion testing using TTCN-3 and [20] provides an analysis of the problem but no actual implementation examples and discussion. In [24], penetration testing is driven by a process that starts with threat modeling.

3 Problem Description

There are two basic tasks in web application testing:

- Specifying test cases, including test actions usually in the form of http requests with test oracles implemented as assertions
- Extracting data from responses written or dynamically generated in HTML format

A number of frameworks [10, 11] are available that can create test cases by simulating user actions in a web browser by among other things executing scripting functions

on the client side. They also have features to reduce the effort of data extraction and requests submission. However, the two above tasks end up being intermingled as shown for example on the HtmlUnit website [10] where the implementation details of extracting the title from the HTML code are intermingled with specifying the expected response that constitutes the test logic.

```
public void testHtmlUnitHomePage() throws Exception {
   final WebClient webClient = new WebClient();
   final URL url = new URL("http://HtmlUnit.sourceforge.net");
   final HtmlPage page = (HtmlPage)webClient.getPage(url);

  assertEquals( "HtmlUnit - Welcome to HtmlUnit", page.getTitleText());
 }
```

Such a strong coupling of data extraction and test assertion tasks (see figure 1a) makes test cases more complex to write and harder to understand. It also makes test specification heavily dependent on the particular tools used for data extraction, which can be problematic if one wishes to migrate to a different tool. In industrial applications we have worked on, we have found that data extraction framework related statements represented about 80% of the total test script source code.

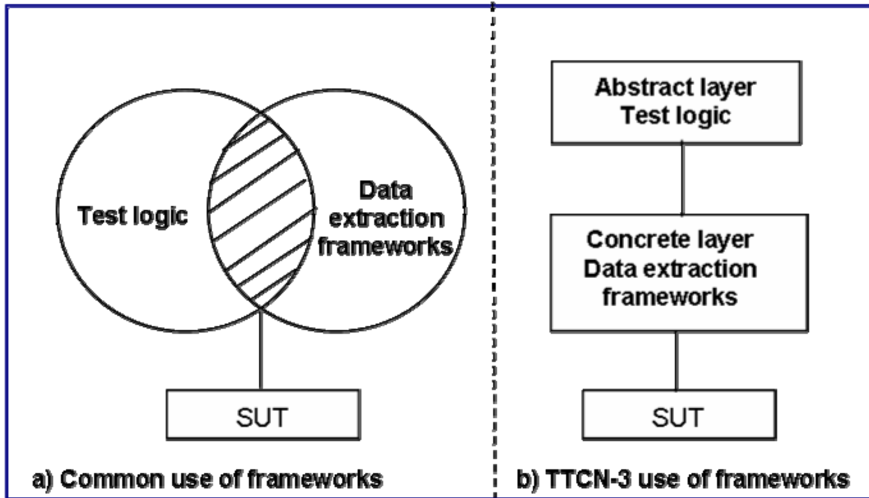


Fig. 1. Use of data extraction frameworks approaches

The use of a test language like TTCN-3 addresses this issue by clearly separating test logic from data extraction logic (see figure 1b). There is an abstract layer where test assertions are specified and a concrete layer that handles communication with the SUT including data extraction. While the abstract layer uses a very powerful matching mechanism that allows composing very complex assertions, the concrete layer can use any general purpose language (GPL) features to perform its tasks, including the type of frameworks used in our example. The significant advantage is that test logic

written in the TTCN-3 abstract layer becomes independent of the framework used for data extraction and communication.

Another important, but perhaps less obvious, advantage of separating test logic from data extraction logic is that it promotes a more efficient programming style by factoring out repeated operations. In theory, one could manually create such factoring using a general purpose language, but with TTCN-3 it is ensured by the TTCN-3 model architecture. As was shown in [23], the first task when testing using TTCN-3 consists of modeling a web page using an abstract data type definition. It consists of mapping each abstract data type to a function in the codec that performs the coding and decoding of concrete data. Consequently, the above example of testing for a title page will result in only a single function that handles the title page extraction.

In the following abstract representation of a web page found in [23]:

```
type record WebPageType {
        integer statusCode,
        charstring title,
        charstring content,
        LinkListType links optional,
        FormSetType forms optional,
        TableSetType tables optional
}
```

The above *WebPageType* data type drives the codec to invoke the concrete *decode-Page*() method that processes a web response and populates an instance of the data type. Each element is processed and for example the title of the web page is obtained using the *getTitleText*() method of the *HtmlUnit* framework *WebClient* class.

```
public RecordValue decodePage(HtmlPage theCurrentPage) {

   RecordValue theResponseValue = (RecordValue)
     typeServer.getTypeForName("HtmlTypes.WebPageType").newInstance();

   …
   String title = theCurrentPage.getTitleText(); // HtmlUnit
   CharstringValue titleValue = (CharstringValue)
                             typeServer.getCharstring().newInstance();
   titleValue.setString(title);
   theResponseValue.setField("title", titleValue);
   …
}
```

The difference with the traditional GPL/frameworks approach is that the above *decodePage()* method is well separated from the test logic because it is located in the concrete layer that is generic in the sense that it is not dependant on a particular web application and thus makes it fully re-usable for any other web application. This is precisely not the case when data extraction and test logic functionalities are intermingled. The additional benefit is that these data extraction functionalities are completely transparent at the TTCN-3 abstract layer. Thus, the additional benefit of using TTCN-3 is to further factor out some code and place it in a framework. The abstract data typing and related codec for web pages inherently constitutes a framework that can be endlessly re-used for various web pages within a web application or for any

new web application as we had already done in [23]. Thus, the only things that remain to be coded by the tester are the assertions that are specified exclusively in the abstract layer using the central TTCN-3 concept of template. Here it is important to stress that, in theory, the test coder could have written a data extraction framework of his own in a GPL, but this is rarely done as there is no built in structured support for it. In TTCN-3 the separation of test logic in an abstract layer from data extraction logic forces the coder to use this more efficient structuring mechanism.

4 Specification of Test Oracles in TTCN-3

Test oracles are specified in TTCN-3 using the concept of a template. A TTCN-3 template is a mechanism for combining multiple assertions into a single operation, thus it is a structured assertion. For example, for web pages, test code written in a traditional general purpose language would use a sequence of independent assertions that will stop at the first failure of one single assertion while TTCN-3 can relate assertions to abstract data types that represent all the essential elements of a web page thus extending the concept of structured data types to the concept of structured assertion. Thus, all assertions composing a template are verified at once and they are all verified whether one fails or not. This gives a full picture of what could be wrong in a given web response. A TTCN-3 template can be best described by comparing it to an XML document with the difference that it contains assertions rather than data. The TTCN-3 matching mechanism also enables one to specify a single structured assertion for an entire web page. This can include complex tables, links or forms using our abstract type for web pages. For example, a test oracle for a web page table can be hard coded as follows. The value assignments (using the ":=" operator) implicitly mean should be equal to, and are therefore an assertion in the traditional JUnit sense:

```
template TableType statements_table_t := {
   rows := {
      {cells := {"date", "description", "amount", "kind"}},
      {cells := {"2009-07-10", "check # 235", "2491.89", "DB"}},
      {cells := {"2009-07-02", "salary ACME", "5000.23", "CR"}},
      {cells := {"2009-06-28", "transfer to savings", "500.0", "DB"}}
   }
}
```

However, hard-coded templates such as this one can require tedious efforts to create and require significant maintenance effort, which makes them not very re-usable.

In the rest of this section we explain the principle of self-definition for test oracles in 4.1, the sharing of test oracles among actors in 4.2 to address parallel execution, and their foundation for reuse in 4.3.

4.1 Test Oracle Self-defining Principle

To illustrate our approach we use a simple penetration testing example that consists of checking if one can illegally login to an application and thus land inside the application that is characterized by a specific web page. This consists in submitting a form filled with the login information and perform an assertion on the content of the specific web page.

We avoid hard coding test oracles by leveraging the concept of template in TTCN-3. The TTCN-3 template is used both for matching incoming data against a test oracle and passing data from the concrete layer to an abstract variable that can be in turn used as a template to be matched against further responses data. In the context of a web application, an abstract response that is the result of one login is used as a test oracle to be matched against the response for another login attempt on the same login form. This approach appears seemingly trivial at the abstract level but relies on a complex infrastructure provided by TTCN-3. It consists of two steps:

- Perform a login using a legitimate user id and password and obtain the normal response page content that we save by assigning it to a TTCN-3 template variable.
- Perform a login using an illegitimate password (SQL injection or stolen from a cookie as in XSS) on the same user id and use the response content from the legitimate login that was stored in a variable as a test oracle against the response from the illegitimate login.

If the legitimate response content of the first login attempt response matches the response to the illegitimate login, there is potential penetration vulnerability. The interesting aspect of this approach is that at no time do we need to explicitly specify the test oracle for the response, thus by definition, no hard coding needs to be performed. As a matter of fact, the tester does not even need to know what the content of the response exactly is. Minimal hard coding could still be used to avoid false positives. For example, checking a return code of 200 [25] or checking the title of the page could increase confidence. This can be easily implemented due to the fact that the template used is now stored as a structured variable where fields can be modified using invariant values such as the 200 return code. All of this is based on the ambiguity between variables that in a GPL can only contain data and that in TTCN-3 can contain templates that are really functions that perform the matching. The template variable allows modifying individual functions rather than just plain data.

Another important difference with traditional testing is that responses are fully assembled in the TTCN-3 concrete layer by the codec that is part of our TTCN-3 framework and thus does not need to be developed for each web application. It can be re-used for any other web application.

The capability of TTCN-3 to assign a complex assertion to a variable and then re-use the variable to actually perform the matching with incoming data is central to our design and has an obvious advantage. In addition to this advantage, the use of a variable as test oracle has one additional advantage; the specified behavior becomes generic and can be used in various web applications without having to rewrite anything. The only code that needs to be rewritten is the login request which is always minimal because it consists in our case only in rewriting the request login template by providing the user id and password and the related form information. This process can be easily automated using TTCN-3. This is the result of TTCN-3's separation of concern between test behavior and conditions governing behavior that are implemented using the TTCN-3 concept of template. In this case, behavior remains constant while conditions change from application to application.

In our example, the test case to test would be divided into the definition of the templates followed by the definition of the test case itself.

First the specification of the legitimate and illegitimate login form templates, data types defined in [23], that themselves use other template definitions:

```
template BrowseFormType legitimateLoginForm_t := {
   name := "",
   formAction :=
            "https://peilos.servebeer.com/Account/Login?ReturnUrl=%2f",
   kindMethod := "post",
   elements := { aems_userId_t, aems_normal_pwd_t, aems_normal_button_t}
}
```

Second, the specification of the illegitimate login form merely re-uses the template definition of the legitimate login form by modifying only the form elements of which only the password element is different since now it consists of the illegitimate string (e.g. SQL injection):

```
template BrowseFormType illegitimateLoginForm_t modifies
                                        legitimateLoginForm_t := {
   elements := { aems_userId_t, aems_sql_injection_pwd_t, aems_normal_button_t}
}
```

Since a TTCN-3 template can re-use other templates at any point, the above specified elements can be specified as separate templates themselves to allow maximal structuring. This is the result of the multiple functionalities of the template that acts both as a variable that can be assigned values or other templates and an implicit function (execution of matching mechanism). For example, the following three templates specify the content of the above form input for a normal legitimate login

```
template FormElementType aems_userId_t := {
        elementType := "text",
        name := "username",
        elementValue := "admin"
}

template FormElementType aems_normal_pwd_t := {
        elementType := "password",
        name := "password",
        elementValue := "123456"
}

template FormElementType aems_normal_button_t := {
        elementType := "submit",
        name := "",
        elementValue := "Log In"
}
```

Now, for an illegitimate login, all we need is to specify a different template for the password element that contains the illegitimate string, in this case, a typical SQL injection value:

```
template FormElementType aems_sql_injection_pwd_t := {
        elementType := "password",
        name := "password",
        elementValue := "' or 1=1 – "
}
```

Finally, the specification of a minimal test case using the above defined templates:

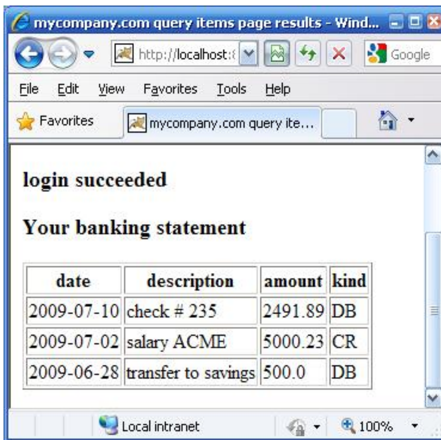
```
testcase AEMS_SQL_injection_attack_with_auto_verification()
                             runs on MTCType system SystemType {
   var template WebPageType legitimateResponse;

   … // open main page

   webPort.send(legitimateLoginForm_t);
   webPort.receive(WebResponseType:?)  -> value legitimateResponse

   … //return to main page

   webPort.send(illegitimateLoginForm_t);
   alt {
     [] webPort.receive(legitimateResponse) {
            setverdict(fail);
     }
     [] webPort.receive(WebResponse: ?) {
            setverdict(pass);
     }

   }

   …
}
```

In the above example, the request template *legitimateLoginForm_t* needs to be changed from application to application only. Each new web application has potentially different input field identifiers which need to be coded accordingly in the login template. The codec in the concrete layer will invoke the appropriate method providing the values of the parameters defined in the abstract template *legitimateLogin-Form_t*. However, it is important to note that while the abstract layer will need to be modified from application to application, the concrete layer codec will not because it handles a generic representation of forms.

So far, this approach may appear to be complex when considering that the same principle could have been achieved with a simple string comparison between the two response contents using any GPL. The reality is not so simple. While in manual testing, a human tester can visually differentiate the content of two web pages on a browser because the rendering eliminates formatting information as shown on figure 2a, in automated testing when using string comparisons, test execution tools can only provide the HTML text in full as for example the following HTML code corresponding to figure 2a.

```
<HTML>
<HEAD>
<TITLE>mycompany.com query items page results</TITLE>
</HEAD>
<BODY >
<h3>login succeeded</h3>
<h3>Your banking statement</h3>
<table border="1" >
<tr> <th>date</th> <th> description </th> <th> amount </th> <th> kind </th></tr>
<tr><td>2009-07-10</td><td>check # 235</td><td>2491.89</td><td>DB</td></tr>
<tr><td>2009-07-02</td><td>salary ACME</td><td>5000.23</td><td>CR</td></tr>
<tr><td>2009-06-28</td><td>transfer to savings</td><td>500.0</td><td>DB</td></tr>
</table>
</BODY>
</HTML>
```

The above HTML code needs to be searched and finding the differences in potentially large HTML code is tedious mostly due to the presence of HTML tags. For example, we have observed that a sign in error web page for a real bank is composed of 258 lines of code and that the actual text indicating that the password entered is invalid is buried deeply at line 85. Instead, TTCN-3 decomposes web responses into structured content eliminating the HTML formatting information as shown on figure 2b in the process and then performs individual comparisons between these content's elements as a human would do, but with the additional benefit of automatically flagging the ones that did not match and thus making them easy to spot. This gives the tester an overview of test results and considerably increases the efficiency of debugging activities by reducing the number of test development iterations.



Data

Name	Value
WebPageType	
statusCode	200
title	mycompany.com query ite...
content	mycompany.com query ite...
links	
forms	
tables	
[0]	
rows	
[0]	
cells	
[0]	date
[1]	description
[2]	amount
[3]	kind
[1]	
cells	
[0]	2009-07-10
[1]	check # 235
[2]	2491.89
[3]	DB

a) Legitimate web response page b) Abstract representation

Fig. 2. Abstract results inspection tools

4.2 Sharing Test Oracles among Actors

TTCN-3 is well known for its capability to compose complex test scenarios that include complex test configurations involving several test actors. In the case of XSS

vulnerabilities, we need to consider at least two actors, the penetration victim and the malicious attacker. Both of these users need to perform operations in a given order as shown on figure 3. Our example illustrates a classic persistent XSS attack that is achieved via a typical bulletin board application where users can post and read messages. A malicious user can post a message containing a script that in our case steals the bulletin board reader's cookies. The malicious user then uses the password stored in the cookie to illegally enter the victim's application. For this test to work, a precise choreography needs to be put in place that in TTCN-3 is implemented using coordination messages. Also, if we want to use the test oracles self-defining principle described in the previous section, we need to be able to make the content of the page reached through a normal login by the penetration victim available to the process that tests that the attacker can also reach that same page by exploiting the XSS vulnerability.

The TTCN-3 concept of parallel test component (PTC) is comparable to the object oriented concept of thread. It is an efficient way to separate the behavior of the two actors. One of the strong points of PTCs is that they also use TTCN-3 ports that can easily be connected at the abstract level without requiring any efforts about communication implementation. In our case, this language feature can be used for two purposes:

- Coordinate the actions of the two actors.
- Pass information from one actor to another.

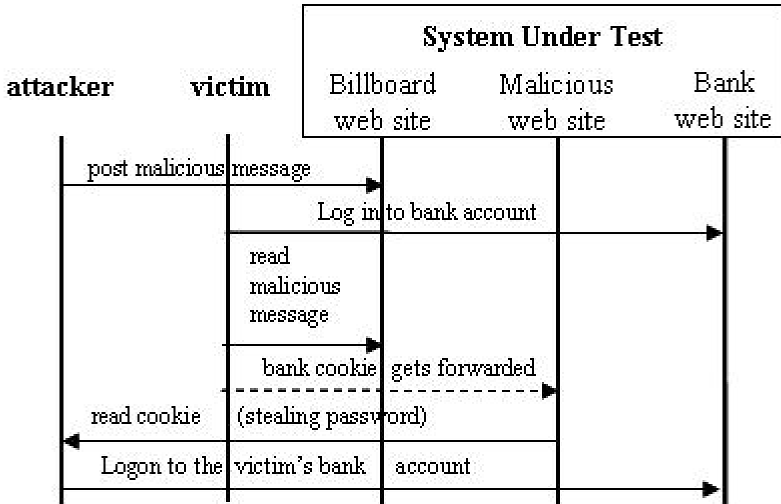


Fig. 3. persistent XSS attack sequence of events

The XSS vulnerability test case consists in three separate pieces of code. The first one describes the test case where two instances of PTCs are created to depict the attacker and victim. The coordination ports and the information passing ports are connected while the communication ports to the web application are mapped to concrete

connections. In our case the concrete connections consist of instances of *WebClient* classes that are part of the concrete layer. Since, the PTCs are independent threads, a number of coordination messages need to be exchanged between the master test component (MTC) and the PTCs as shown in the following test case:

```
testcase XSS_PersistentAttack() runs on MTCType system SystemType {
   var PTCType attacker := PTCType.create("attacker");
   var PTCType victim := PTCType.create("victim");

   connect(mtc.attackerCoordPort, attacker.coordPort);
   connect(mtc.victimCoordPort, victim.coordPort);
   connect(victim.infoPassingPort, attacker.infoPassingPort);

   map(attacker:webPort, system:system_webPort_attacker);
   map(victim:webPort, system:system_webPort_victim);

   attacker.start(attackerBehavior());
   victim.start(victimBehavior());

   attackerCoordPort.send("post message on bulletin board");
   attackerCoordPort.receive("post message done");
   victimCoordPort.send("login ");
   victimCoordPort.receive("login performed");
   victimCoordPort.send("read bulltein board");
   victimCoordPort.receive("read bulletin board done");
   attackerCoordPort.send("login");
   attackerCoordPort.receive("login performed");

   all component.done;
}
```

The discovered test oracle on the victim's PTC must now be passed to the Attacker PTC since they are running on different threads. This is achieved merely by the victim's PTC sending a message to the attacker's PTC using the tester defined *infoPassing* port in the respective behavior functions of the victim and the attacker that are executed in parallel as shown on figure 4.

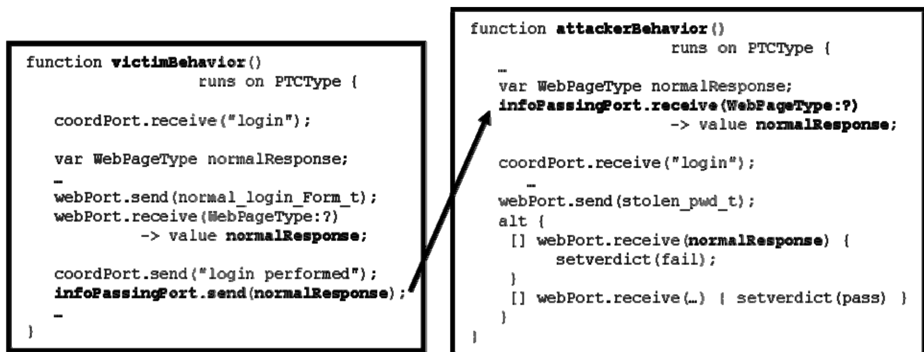


Fig. 4. Inter process transmission of test oracles

In the separate processes shown on figure 4, we use the blocking receive statement to control the execution of the attacker behavior rather than using a coordination message.

4.3 Re-usability of the Test Suite

Our test system is composed of a large portion of re-usable code at various levels. Besides the re-usability of the test oracle self-defining principle, the infrastructure on the SUT side is also re-usable for various different applications. For example, bulletin board message posting and cookie stealing mechanisms remains unchanged from application to application. Thus, the only portion that needs to be modified from application to application is confined to the login templates at the abstract layer and the parsing of cookies to actually steal passwords at the malicious attacker's cookie gathering application. It seems likely that penetration test patterns could be defined for these with TTCN-3 templates used as examples.

5 A Proposed Refinement to the TTCN-3 Model Architecture

So far we have discussed the advantages of the TTCN-3's separate abstract and concrete layers without providing too much detail on the concrete layer side. The concrete layer has two basic functionalities:

- A codec that translates the internal abstract representation of data to and from bytes.
- A test adapter that sends and receives data to and from the SUT as bytes.

The TTCN-3 standard part V [6] and part VI [7] clearly defines two interfaces for these two functionalities, namely the *TciCDProvided* interface and the *TriCommunicationSA* and TriPlatformPA interfaces respectively. Both need to be implemented by the test application developer to create a concrete codec and a concrete test adapter. Because TTCN was originally conceived for telecommunications applications that consist mostly in sending and receiving compacted information as bytes, the architecture of TTCN-3 execution tools consists of a strict sequence of invoking coding or decoding methods of the implemented codec class to obtain bytes and then invoking the *triSend* method of the implemented test adapter class in order to concretely transmit the bytes to the SUT over a communication media and vice versa as shown on figure 5a. This architecture is not usable when using object oriented frameworks such as *HtmlUnit* mostly because with these frameworks there is no such a clear distinction between data coding/decoding and data transmission as in the TTCN-3 model but also because of its original byte stream orientation it has not been designed for keeping states of objects as required by web testing frameworks. In short, the two TTCN-3 concrete layer classes can not be mapped directly to the single *HtmlUnit WebClient* class. For example, with *HtmlUnit* we have only one class, *WebClient*, instead of two to accomplish both codec and test adapter functionalities. The obvious solution to this mismatch of architectures would consist in bypassing the TTCN-3 codec class altogether and having *WebClient* object instances residing in the TTCN-3 test adapter. This solution requires a modification of the TTCN-3 execution tools

architecture. The TTCN-3 codec can not be eliminated because of the requirements of telecommunication applications but it can be adapted so as to act merely as a relay that passes abstract value objects down to the test adapter where frameworks can be used to both perform codec and data transmission functionalities as shown on figure 5b. Thus, we have created an *HtmlUnit WebClient* driven codec and test adapter. For this to work, the abstract values from the abstract layer must reach the test adapter without being converted to bytes by the traditional codec. This is actually easy to implement because even with the traditional architecture, the messages coming from the codec would not arrive directly as bytes but as objects instances of the class implementation of the *TriMessage* interface also defined in [7] that contain an attribute for the bytes. Thus, the solution would be to extend the *TriMessage* implementation class with an attribute containing the abstract values. The problem with this solution is that the standard specifies that the implementation of the *TriMessage* interface is the responsibility of the tool provider and usually this is proprietary. Thus, TTCN-3 execution tools are written so as to process *TriMessage* implementation objects but not any user defined extensions. Thus, this solution requires the collaboration of tool vendors which we obtained and it is to be noted that this solution does not require any changes to the TTCN-3 standard since *TriMessage* is an interface only. However, we do recommend that this feature be implemented in the TTCN-3 standard so as to promote this efficient solution to external frameworks integration.

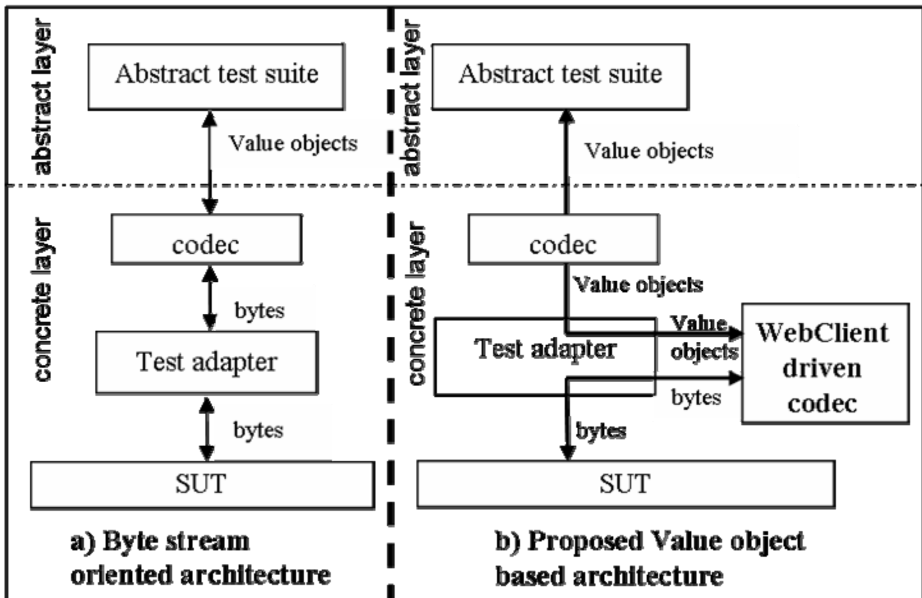


Fig. 5. TTCN-3 architectural models

This new architecture is a considerable improvement over previous work around such as making *WebClient* objects available to the codec class instance either through the user extended codec class constructor or via serialization which is not always

possible because objects need to implement the *Seriazible* class which is not even the case for the *WebClient* class. For example, a form submission would consist in invoking the *click()* method of the *Button* object of the login form. This clicking a button functionality makes no sense to be modeled using a byte stream. This requires populating the *Form* object instance with the parameter values that arrive as a TTCN-3 abstract *Value* object from the abstract layer. This *click()* method actually sends the request using the HTTP connection of the *WebClient* object. This architecture is even more important when there are client side javascript functions to be executed. In this case, even the concept of the SUT is no longer as clear as the TTCN-3 model defines it. Our approach proves to be particularly efficient in case of multiple user configurations using PTCs.

6 Conclusions

In this paper, we have shown how TTCN-3 can be effectively combined with frameworks like HttpUnit and HtmlUnit. Up until now, there has been little awareness in the literature that TTCN-3, as a telecom standard, was appropriate for application penetration testing, and certainly no discussion on how best to use TTCN-3. Of course, using a test specification language like TTCN-3 requires more sophistication and training of testers in order to use the tool effectively, but given the high cost of software development and the critical nature of web application security, we believe such an investment is more than worthwhile.

TTCN-3 provides a more systematic and effective approach to penetration testing by

- Separating test logic from data extraction logic.
- Leveraging test oracles and templates for more powerful assertion writing, support of parallel execution tests, and reuse.

These were illustrated with examples based on our experiences in actual projects in industry. However, more systematic case studies need to be done to validate and quantify the benefits that can be achieved by our approach.

It would also be beneficial to investigate the systematic create of test oracle templates, and penetration test patterns to address common situations that occur in web application penetration testing.

Finally, we have identified a key limitation in the current TTCN-3 model architecture as defined by the standard and proposed a simple refinement of the architecture to address it. A TTCN-3 tool vendor has already incorporated the change in their most recent tool offering.

Acknowledgements

The authors would like to thank Testing Technologies IST GmbH for providing us the necessary tool -- TTworkbench to carry out this research as well as NSERC for partially funding this work.

References

1. Andreu, A.: Professional Pen Testing for Web Applications. Wrox Press (2006)
2. Arkin, B., Stender, S., McGraw, G.: Software Penetration Testing. IEEE Security & Privacy 3(1), 84–87 (2005)
3. AttackAPI (2010), http://www.gnucitizen.org/blog/attackapi/ (retrieved November 2010)
4. Bishop, M.: About Penetration Testing. IEEE Security & Privacy 5(6), 84–87 (2007)
5. ETSI ES 201 873-1 (2008). The Testing and Test Control Notation version 3, Part 1: TTCN-3 Core notation, V3.4.1 (September 2008)
6. ETSI ES 201 873-5 (2008). The Testing and Test Control Notation version 3, Part 5: TTCN-3 Runtime Interface, V3.4.1 (September 2008)
7. ETSI ES 201 873-6 (2008). The Testing and Test Control Notation version 3, Part 6: TTCN-3 Control Interface (TCI), V3.4.1 (September 2008)
8. FireBug (2010), http://getfirebug.com/ (retrieved November 2010)
9. GreaseMonkey (2010), https://addons.mozilla.org/en-US/firefox/addon/748/ (retrieved November 2010)
10. HtmlUnit (2010), http://HtmlUnit.sourceforge.net/ (retrieved November 2010)
11. HttpUnit (2010), http://HttpUnit.sourceforge.net/ (retrieved November 2010)
12. JUnit (2010), http://www.junit.org/ (retrieved November 2010)
13. Brzezinski, K.M.: Intrusion Detection as Passive Testing: Linguistic Support with TTCN-3 (Extended Abstract). In: Hämmerli, B.M., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 79–88. Springer, Heidelberg (2007)
14. Manzuik, S., Gold, A., Gatford, C.: Network Security Assessment: From Vulnerability to Patch. Syngress Publishing (2007)
15. Metasploit project (2010), http://www.metasploit.com/ (retrieved November 2010)
16. OWASP Testing Guide, OWASP Testing Guide (2008), https://www.owasp.org/images/8/89/OWASP_Testing_Guide_V3.pdf (retrieved November 2010)
17. OWASP TOP 10, OWASP TOP 10: The Ten Most Critical Web Application Security Vulnerabilities (2007), http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf (retrieved November 2010)
18. Palmer, S.: Web Application Vulnerabilities: Detect, Exploit, Prevent. Syngress Publishing (2007)
19. Potter, B., McGraw, G.: Software Security Testing. IEEE Computer Society Press 2(5), 81–85 (2004)
20. Prabhakar, T.V., Krishna, G., Garge, S.: Telecom equipment assurance testing, a T3UC India presentation (2010), http://www.ttcn3.org/TTCN3UC_INDIA2009/Presentation/1-ttcn3-user-conference-nov_updated_-2009.pdf (retrieved November 2010)
21. SANS TOP 20, 2010 TOP 20 Internet Security Problems, Threats and Risks, from The SANS (SysAdmin, Audit, Network, Security) Institute (2010), http://www.sans.org/top20/ (retrieved November 2010)

22. Splaine, S.: Testing Web Security: Assessing the Security of Web Sites and Applications. John Wiley & Sons, Chichester (2002)
23. Stepien, B., Peyton, L., Xiong, P.: Framework Testing of Web Applications using TTCN-3. International Journal on Software Tools for Technology Transfer 10(4), 371–381 (2008)
24. Thompson, H.: Application Penetration Testing. IEEE Computer Society Press 3(1), 66–69 (2005)
25. Xiong, P., Stepien, B., Peyton, L.: Model-based Penetration Test Framework for Web Applications Using TTCN-3. In: Proceedings Mce.Tech. 2009. Springer, Heidelberg (2009)