Using TTCN-3 as a Modeling Language for Web Penetration Testing

Bernard Stepien, Liam Peyton, Pulei Xiong

SITE, University of Ottawa, {bernard, lpeyton, xiong}@site.uottawa.ca

Abstract—**Penetration testing is widely used for vulnerability assessment of web applications. Usually, it is performed by specialized security experts after development is completed and the application deployed into production, but recent research has proposed a model based penetration test framework for web applications which provides a repeatable, systematic and cost-efficient approach fully integrated into a security-oriented software development life cycle. In this context, we evaluate the test specification language TTCN-3 as a modeling language for web penetration testing and show how its inherent abstraction features make the process of generating web penetration test campaigns easier. In particular, we demonstrate the advantages of combining separate models for the relevant web vulnerabilities and web application functionalities, with a generic web abstraction model and a TTCN-3 test framework model.**

Index Terms—**Modeling, Penetration Testing, Security, Software Engineering, Web Security, TTCN-3**

I. INTRODUCTION

Penetration testing is widely used in industry as a method for security assessment [18] to identify vulnerabilities in web applications [19]. It is often performed by security experts as a post-deployment, isolated test task to measure an application's security posture. It has been proposed by many researchers that penetration testing should be leveraged for security assurance when an application is still under development, and it is well recognized that the testing should be performed early in the software development life cycle so that any security defects can be fixed with less cost [27], [28]. This is especially true for those vulnerabilities that are introduced by deficient analysis [25], [26].

Unfortunately, penetration testing is usually performed very late in the lifecycle of a web application by security experts in an ad hoc manner with limited tool support [5]. A model-driven penetration test framework for penetration testing web applications has been proposed in [15] that integrates penetration testing into a security-oriented software development lifecycle by deriving test scripts from development artifacts. In this paper, we present an approach that leverages the inherent abstraction features of the test specification language TTCN-3 to make the process of generating web penetration test campaigns easier. In particular, we demonstrate the advantages of combining separate models of web vulnerabilities and web application functionalities, with a generic web abstraction model and TTCN-3 test framework model.

II. BACKGROUND

Model–based web testing has been widely studied. It is a software testing approach based on generating test cases from models. It provides an oracle to determine test results based on models that describe an application's expected behaviors [20]. A taxonomy of model-based testing can be found in [13]. It considers model-based testing as an exercise in abstraction at the following levels:

- function abstraction
- data abstraction
- communication abstraction

The characteristics of a test system are also important: deterministic or non-deterministic, timing issues, continuous or event discrete models and the use of separate models for different testing purposes.

We propose an approach that uses a high-level language, TTCN-3 [6] that was specifically designed for specifying and executing test suites at an abstract level and which gives full control of all these aspects.

Other approaches include:

- Finite State Machines.
- state-based (or pre-post) notations
- transition-based notations
- history-based notations
- functional notations
- operational notations
- stochastic notations
- data-flow notations

A model consists in describing the sequences and nature of inputs and outputs of a system under test (SUT) that constitute the test behavior. Such behaviors are specified using some formal languages that can be executed to produce test cases automatically. The most common paradigm is to use some finite state machine (FSM) from which all possible sequences of events (input or outputs) can be automatically generated. The major problem to solve is the one of state explosion when FSMs are not finite. This problem is most commonly solved by constraining the test case generation to particular test purposes that describe how to reach a given state possibly using specific paths or way points. Many different formal and informal languages have been used to describe models. Telecommunications engineers used early formal languages such as LOTOS [8] or SDL [9]. With the arrival of object oriented languages, UML [10] became one the specification languages of choice, although it lacks a consistent, operational semantics [24].

III. Web Penetration testing using TTCN-3

A. Differences between functional testing and penetration testing of web applications

Before defining models to support web penetration testing we need to note the main differences between model-based functional testing of web applications and model-based penetration testing. Testing of non-functional requirements such as security and usability presents challenges not always encountered in functional testing. There is still some question as to the advantages of model-based testing for non-functional requirements (including security) [17].

Functional testing of web applications starts from scratch. A full web application functional model must be coded using a formal language and test cases are generated from it.

Web penetration testing, on the other hand, is usually achieved in two steps. The first step consists in using a fully functional web system for which a web application functional model already exists, and for which functional test cases have already been generated and executed. The test execution traces are usually saved into log files. The web application does not need to be deployed and even can be in various stages of completion at various stages of the web application development cycle [12].

The second step consists in specifying a web penetration test model in order to determine where in an existing sequence of web events obtained from the first step of functional testing, specific penetration testing events must be inserted or the original test suite must be modified for the purpose of penetration testing.

Thus, web penetration testing does not need to be achieved from a complete model that describes both web application functionality and security requirements but instead is derived from a combination of the observed behavior of a fully functioning web application and a model of web vulnerabilities. Thus, in this paper, we assume that the web application functional model that has been used to generate web functional test cases has already been established using one of the numerous methods found in literature and that testing traces in form of log files also already exist and can be used for the second step of penetration testing.

B. Combined models

Consequently, there needs to be two separate models available in order to generate web penetration test suites, one for the functional aspect of the web application and one for the web penetration aspect. However, we have determined that when using TTCN-3, the problem can be further divided into test software components that each has its own model. Thus, in this project, we have determined that there are two additional models that are important as shown on figure 1. First, the TTCN-3 language as a testing language has its own model that is based on the concepts of separation of concerns between an abstract and concrete layer and within the abstract layer there is also a separation of concern between behavior and conditions governing behavior revolving around the TTCN-3 concept of template. This model addresses one of the abstraction requirements of any class of models. Second, we have constructed a specific abstraction model using the TTCN-3 language itself to describe web testing using abstract data types that separate application data at an abstract level from HTML implementation and encoding details [16].

The interesting aspect of these four overlapping models is that two of them are specified using model specification languages while the two others are inherent to the test implementation language, in this case TTCN-3 but also that TTCN-3 enables us to shift some elements of the functional and penetration models to the test case implementation itself.

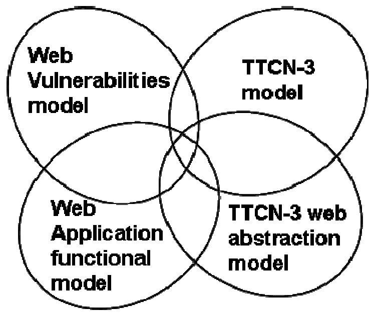

Web Vulnerabilities model
TTCN-3 model
Web Application functional model
TTCN-3 web abstraction model

Figure 1: Models leveraged in web penetration testing

It was also established in [11] that it is an advantage to break down models into sub-models or views. They use the following architecture viewpoints: functional, logical, technical and topological views in modeling automotive systems. Since the penetration test campaigns are conducted upon a well-structured model, automation of major steps in the test process becomes possible.

IV. Preliminary experiments

We have made two sets of preliminary experiments involving different case studies in our previous work. The first one used general purpose languages combined with web testing frameworks [15] and the second one used the TTCN-3 language also combined with web testing frameworks [16].

A. Generating test cases from vulnerabilities databases

In our first experiment [15], a knowledge base of known web vulnerabilities was created based on a model of vulnerabilities, attack vectors (types of entry points) and fuzz vectors (variations on parameters that can be used in the attack). It was populated from widely used Internet sources [1][2][3][4]. Then, a single penetration test campaign model, shown in figure 2, was used to model how a specific web application should be tested for web vulnerabilities by modeling how the application footprint, entry points, and check points mapped to known vulnerabilities and test cases (defined in terms of attack vectors and fuzz vectors). An entry point is an interface point (usually identified by an URL) where requests can be formulated and sent to the web application, thus serving as potential "doors" to web application attacks. Check points define where expected results can be used to verify that appropriate protection has been provided in the web application. An application footprint consists of the elements of the running environment elements such as platform, backend database, OS and encoding schema. The application footprint is used to filter out those attack vectors and fuzz vectors that are not applicable to the specific running environment of a web application under test.

To evaluate the approach, in that first experiment, a web application penetration test model was created for the WebGoat reference application [23] and test cases were generated from the model and a test campaign was run. The test cases were generated as HTTP request scripts that were run by a hard-coded test runner tool built specifically for our experiment. Similar test campaigns were also run against two healthcare web applications [21] [22] while they were under development before being deployed as pilot systems.

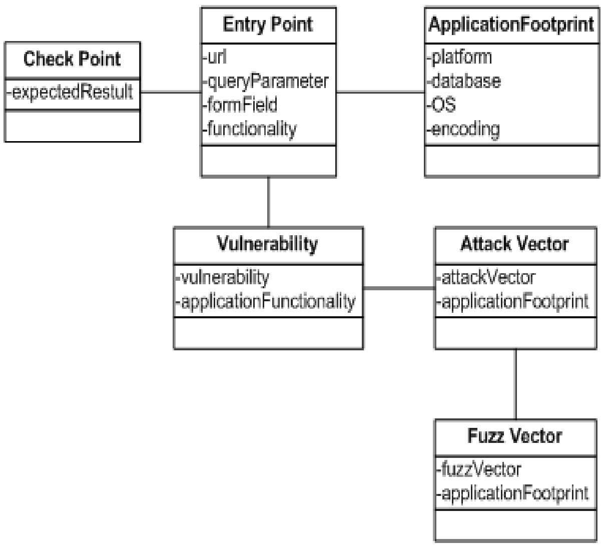


Figure 2: Web application penetration test security model

This experiment enabled us to determine how to correlate information in vulnerabilities knowledge bases such as OWASP [1] to test case requirements and application functionalities. It also showed that the approach was successful in finding vulnerabilities that existed in the three web applications. While most of the process could be automated, some areas still needed to be hand coded. The most important finding of this experiment was that there was little need for finite state machines to specify penetration tests. In this approach, most of the effort to determine where to apply knowledgebase of known web vulnerabilities was either spent gathering information from the web application developers or it was spent performing web-crawling to determine the entry points and implementing the attack vectors. This is mostly due to the fact that web penetration is deterministic, thus, it can be represented by a simple sequence of test events.

B. Generating TTCN-3 test cases

In our second experiment, we took the test scripts generated in the first experiment and evaluated the use of TTCN-3 as a specification language for those test scripts. This experiment enabled us to determine the mapping between vulnerabilities knowledge bases information and TTCN-3 artifacts such as templates, parallel test components (PTCs) and test behavior paradigms like sequences and alternatives of test events. Four main findings came out of this process:

- how to abstract data via TTCN-3 templates
- how to structure the architecture of parallel test components required for various attack scenarios
- how to specify behavior of attacker and victim
- how to coordinate the behaviors of the attacker and the victim

1) Deriving penetration tests from functional web test cases and vulnerabilities data bases

For a given application footprint, the fundamental element of a vulnerability is its entry point in a sequence of web events. Thus, in the simplest case, one needs to determine such a test event and to modify it according to the fuzz vector provided by the vulnerabilities data base. The attack vector enables us to determine what other test events, either performed by the victim or the attacker, must be inserted in the existing functional test event sequence; how they will be distributed among parallel test components (PTCs); and how they will be coordinated. We have used message sequence charts (MSC) to specify the test events related to a specific vulnerability as shown in figure 3 for the case of the persistent cross-site scripting (XSS) attacks. The vulnerability data base is written in an informal language. Our first task was to manually translate these into a more formal expression using the TTCN-3 language itself.

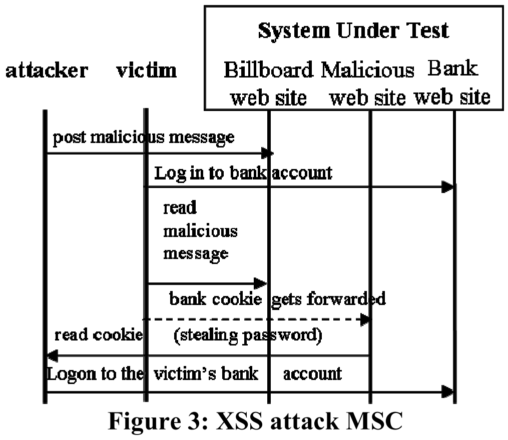


Figure 3: XSS attack MSC

2) Using TTCN-3 test logs to generate web penetration tests

One of the characteristics of TTCN-3 tools is that they produce log files in various form, one of them, in the case of TTworkBench [7] is a textual log file that displays the content of sent and received messages in a TTCN-3 template format. This enables one to extract these templates either manually or automatically since the log has clearly defined keywords, assign names to them and use them for replay, including replay modified with fuzz vector data. In addition to replay, these templates can be used to determine which events constitute the target of attack vectors, such as login pages and which responses constitute check points such as viewing a bank statement web page in an online banking application.

One of the main advantages to working with a trace log represented using TTCN-3 templates is that it is already an abstract representation of the content and consequently it is easier to automate the search using the entry point criteria from the penetration model. For example, in our case, both SQL injections and XSS attacks use manipulations of passwords to penetrate a web application. Thus, the entry point is a test event that consists in submitting a form that contains a password element type. This entry point however is subject to two different kinds of manipulations depending on the attack vector:

- for SQL injection, it merely consists in creating a new form submit template where the password is replaced by the SQL injection fuzz vector using a test behavior identical to the victim's normal behavior
- for XSS, the password is obtained through stealing cookies from the victims session and using it to submit the login form. In this second case, the login test sequence is executed twice, once by the victim and once by the attacker, thus resulting in a more complex test behavior. In this case also, as shown on figure 3, there are a number of other attack vector events to perform

Thus, in the sequence of web penetration events, some events must be marked as entry points. This will result in any penetration web events preceding the entry point to be executed before the entry point event itself. It will also result in any penetration web events after the entry point to be executed after the entry point execution in the functional test sequence. For example, in figure 3, the entry point is the user login event. Thus, the web penetration events of posting a malicious message on a billboard would be executed before the victim's login event. All the remaining events of the victim, including reading the malicious message results, the attacker reading the malicious site to steal the victim's identity information via the cookie and finally the login to the victim's online banking service would be executed after the entry point. Also, the victim's login event following the entry point is used as a check point to confirm the attacker's successful web application penetration.

V. SPECIFYING THE MODELS FOR WEB PENETRATION TESTING

One question to consider is whether the TTCN-3 language is to be used only as a test case implementation language or whether it can also be used as a model language for web penetration testing. In [17], the general purpose C# programming language was used as a model language. Their models are compiled in an intermediary language. In our case, we specify the models for web penetration testing in TTCN-3 directly and do not need to compile it into an intermediary language like in [17]. This is mostly because templates in combination with the TTCN-3 matching mechanism can be used in order to search the test behaviors of existing functional testing test cases that have already been generated from web functional models in a very efficient and more intuitive manner. This is easier to use than C# which has no matching mechanism and a more cumbersome approach to templates. The TTCN-3 matching mechanism combined with templates naturally hides any complex logic that is represented in C# using if-then-else constructs as shown in the examples provided in [17] and thus enables the reader to focus on behaviour.

One advantage of using TTCN-3 is that the models are by definition abstract. Consequently, this consists in using TTCN-3 for describing the web vulnerabilities model of figure 1. Thus, three out of the four models shown on figure 1 are now using TTCN-3. However, the remaining fourth model, the web application functional model is somewhat related to TTCN-3 as well since it can be used itself to generate test cases expressed in TTCN-3. These TTCN-3 test cases can be executed and the execution logs will be one of our central data to generate the web penetration test cases.

The test case generation process is achieved in three steps:

- Transform TTCN-3 test case execution log files into ordered lists of structured templates consisting of an event kind field, a test component identifier and the actual event data transformed into a template. We call this the functional test log events list.
- Specify the web penetration attack vector in a similar way using a similar structured data type but first augmented with a field indicating whether the event is an entry point or a check point and also by setting most of the information of templates to the any value or the any or omitted value values.
- Process the functional test log events list using the TTCN-3 match construct to attempt to match the functional test log events against one of the entry points of the web penetration model events. When a match is found, insert the web penetration events in appropriate test component behaviors and generate the coordination messages between the master test component (MTC) and the various PTCs (victim and attacker PTC).

A. Using the TTCN-3 language as a model language

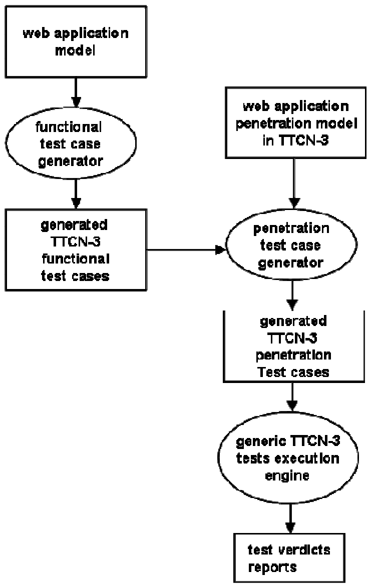
The above procedure implies that we in fact are using the TTCN-3 language as the model language for web penetration testing. There are several advantages in this approach:

- Test events specified as TTCN-3 templates can be modified to transform functional test events into penetration test events. For example, for SQL injections, one simply replaces the password value in the form submitting event for the login with the SQL injection value taken from the fuzz vector.
- Searching and merging functional test events and penetration test events takes full advantage of the TTCN-3 *match()* built-in function to compare a

template and a value, one for the functional test event (FT_event_i below) and the other for the penetration test event (PT_event_j) as follows:

```
match(FT_event_i, PT_event_j)
```

Also, using TTCN-3 as a model language meets at least one of the fundamental requirements of a model language and this is to be abstract. The test generation architecture using TTCN-3 is summarized in figure 4.



web application model
functional test case generator
generated TTCN-3 functional test cases
web application penetration model in TTCN-3
penetration test case generator
generated TTCN-3 penetration Test cases
generic TTCN-3 tests execution engine
test verdicts reports

Figure 4: Penetration testing architecture

B. Implementation details

1) Abstract data typing requirements

As we have already mentioned previously, we use the log files of the TTworkbench tool in order to create TTCN-3 templates to portray the functional test events. These log files describe the test events structured in a number of fields but more important in an abstract way making it easy to manipulate. The Log Event Type field tells us whether the message has been sent or received. The *msgValue* field is of particular interest to us because this one is formatted like a TTCN-3 template. This *msgValue* can be copied and pasted directly into a template definition by the test case generator. The log file for a login sequence looks as follows:

```
Log Event Type : TLI:tliMSend_m
msgValue : HtmlTypes.Url:
"http://localhost:8080/ePasswordsCheck/servlet/query
_pwds"

Log Event Type : TLI:tliMReceive_m
msgValue : HtmlTypes.WebPageType: {
    statusCode := 200,
    title := "mycompany.com login page",
    content := "mycompany.com login page
Bernard's Bank
login
enter your user id and password
user id:
Password
login",
    links := {},
    forms := {
       { name := "loginForm",
        formAction :=
           "/ePasswordsCheck/servlet/pwd_check",
        kindMethod := "post",
        elements := {
         {elementType := "text", name := "userid",
          elementValue := ""}}
       }
    },
    tables := {}
}

Log Event Type : TLI:tliMSend_m
msgValue : HtmlTypes.BrowseFormType: {
 name := "loginForm",
 formAction
:="http://localhost:8080/ePasswordsCheck/servlet/pwd
_check",
 kindMethod := "POST",
 elements := {
    {elementType := "text", name := "userid",
     elementValue := "bernard"},
    {elementType := "password", name := "pwd",
     elementValue := "ottawa"},
    {elementType := "submit",name := "ignore",
     elementValue := "login"}
 }
}

Log Event Type : TLI:tliMReceive_m
   msgValue : HtmlTypes.WebPageType: {
     statusCode := 200,
     title := "MyBank.com chequing account
statement",
     content := "MyBank.com chequing account
statement
login succeeded
Your chequing account statement
date description    amount   kind
2009-07-10 check # 235  2491.89  DB
2009-07-02 salary ACME  5000.23  CR
2009-06-28 transfer to savings  500.0  DB",
  links := {},
  forms := {},
  tables := {
   {
    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"}}
    }
   }
  }
}
```

Thus, in the above log, the first event is a URL submission event. After extracting the template expression found as the msgValue in the log file, we can assign a name to it such as:

```
template UrlType event_1 :=
     "http://localhost:8080/.../query_pwds";
```

This template name can be reused to build new templates or to build elements of more complex templates. Also note that the data type of the event can be extracted from the log (not shown in the above example).

In our previous work [14], we have established a TTCN-3 abstract data typing system to represent web test data. Three basic types were determined: a URL submission type, a form submission type and a web pages type. All of these abstract web events by eliminating any reference to the HTML formatting code and instead focusing on content and purpose. For example, the data type to represent web pages is:

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

This constitutes our web abstraction model. However, for our test case generation purpose, we did wrap these data types into an test event description data type. The data type indicates whether the test event is sending or receiving messages, the name of the test component where the test event occurs, and finally the nature of the event message as a union between several possible data types. All of this data is drawn from the actual log file. The required data types are as follow:

```
 type union WebEventKindType {
    UrlType url,
    BrowseFormType browseForm,
    WebPageType webPage
 }

type record WebEventType {
  charstring action_,
  charstring testComponent,
  WebEventKindType event
}
```

Thus, the functional test event templates depicting the URL submission event and the login form submitting event of the log file would look as follows:

```
template WebEventType functionalTestEvent_1 := {
  action_ := "send",
  testComponent := "victim",
  event := {url := event_1}
}

template WebEventType functionalTestEvent_3 := {
  action_ := "send",
  testComponent := "victim",
  event := { browseForm:= event_3}
}
```

In the above example, note the TTCN-3 syntax that requires indicating the variant of the union type used for the event data. This is particularly important when the matching mechanism is in action as it will always first check the data types of the compared objects before even attempting to compare the actual values.

The web penetration testing model events are specified in a similar way but with an additional field (*modelEventKind*) that indicates whether the event is an entry point or, check point or a regular event.

```
type record ModelEventType {
    charstring action_,
    charstring testComponent,
    charstring modelEventKind optional,
    WebEventKindType event
}
```

2) *Performing the entry point search*

The functional test events are real values since they come from a test execution log file even though we have transformed them into templates. The transformation into templates is done mostly in order to be able to modify them using the full re-usability provided by TTCN-3 templates. We use the TTCN-3 *modifies* language construct for redefining templates. It states that a new template is a modification of an already existing template constitutes. It is in fact a distinct concept of class instance inheritance rather than the traditional class inheritance. We are using this language feature to transform functional test event templates into attack events by merely specifying the delta, i.e. the fields that contain the fuzz vector values.

The penetration test model events are templates in the real sense, i.e. not really values but more also using matching rules. Effectively, when we are searching the functional test events for a form submission that contains a form with an element of type password, we do not need to know all the details of the form submission at that stage. Thus, the template to describe the entry point test event for a login form and a check point for a web page would be as follows:

```
template ModelEventSequenceType
            penetrationModelTestSequence_t := {
  {
    action_ := "send",
    testComponent := "victim",
    modelEventKind := "entry point",
    event := {browseForm := maliciousForm}
  },
  {
    action_ := "receive",
    testComponent := "victim",
    modelEventKind := "check point",
    event := { webPage := ? }
  }
}
```

Where the *maliciousForm* template is described as follows:

```
template BrowseFormType maliciousForm := {
  name := ?,
  formAction := ?,
  kindMethod := "POST",
  elements := {
    {elementType := "text", name := ?,
     elementValue := ?},
    {elementType := "password", name := ?,
     elementValue := ?},
    {elementType := "submit", name := ?,
     elementValue := ?}
    }
}
```

The above form is constituted quasi exclusively of any value symbols "?" except for element types values, especially for the password. This template will match only against forms events from the functional test events and also only forms that contain a password input field, i.e. for example, this means that it will never match against some other data submission form that do not contain a password field.

However, the above template is used only for locating entry points and check points in the penetration testing behavior model. As a first step, once these points are determined, a variable of type *WebEventType* that we name *entryPoint* is constructed by merely eliminating the *modelEventKind* field information. Now, the search can be achieved as a list look up using the TTCN-3 *match* construct.

```
for(i:=0; i < sizeof(functionalTestSequence_t);
                                          i := i +1) {
  if(match(valueof(functionalTestSequence_t[i]),
                   entryPoint)) {
       entry_point_i := i;
  }
  … // similar procedure for check points
}
```

3) Handling check points

Since log traces are simple sequences of events, a simple substitution of a test event by a modified penetration test event is not sufficient. Effectively, the checkpoint event no longer can be a single event. It is now associated with various test verdicts depending on whether or not this test event matches or there is a time out. This requires the specification of alternative behavior using the *alt* construct as follows:

```
alt {
  [] attackerPort.receive(check_point_event) {
        setverdict(fail);
  }
  [] attackerPort.receive {setverdict(pass)}
  [] myTimer.timeout {setverdict(inconc)}
}
```

In the above alternative, the verdict has been set to fail when the attack manages to reach the check point web page. For any other web page, we consider the test to have passed since the attacker was not able to access the protected web page that constitutes the check point.

4) Generating test behavior

Once the position of the entry point in the functional test events list is determined, we can proceed to the last step that consists of merging the functional test events list with the web penetration model test events list. Two possibilities exist depending on the attack vector kind:

- The functional test entry point event template is merely modified and the sequence as such is not modified. This is the case where the attacker substitutes itself for the victim such as for SQL injections attacks.
- Both the sequence and the content (templates) of the functional test events sequence are modified. This is the case for XSS attacks where the attacker lures the victim onto a web site in order to steal his credentials and then in a later stage reproduces the normal behavior of the victim using the stolen credentials.

In both case, the entry point events are modified by the attacker, and we re-use the check points of the victim's functional test events to validate the resulting behavior of the attacker. The TTCN-3 template *modifies* language construct is particularly efficient for this purpose. In our example, for an SQL injection, all that is needed is to modify the third functional test event by inserting fuzz vector data "`' or 1=1 --`" without having to redefine all the other field values:

```
template BrowseFormType attack_event_3
                        modifies event_3 := {
  elements := {
    {elementType := "text", name := "userid",
                 elementValue := "bernard"},
    {elementType := "password", name := "pwd",
                 elementValue := " ' or 1=1 -- "},
    {elementType := "submit", name := "ignore",
                 elementValue := "login"}
  }
}
```

The process of generating the penetration test sequence consists in traversing the functional test sequence and either keeping an event as is or replace it using an attack event as follows:

```
for(i:=0; i < sizeof(functionalTestSequence_t);
                                          i := i +1) {
  if(match(valueof(functionalTestSequence_t[i]),
                                    entryPoint)) {
     entry_point_i := i;
     penetrationTestSequence[j].action_ :=
               functionalTestSequence_t[i].action_;
     penetrationTestSequence[j].testComponent :=
          functionalTestSequence_t[i].testComponent;
     penetrationTestSequence[j].event :=
         substituteValues(entryPoint,
                 functionalTestSequence_t[i].event);
     j := j + 1;
  }
  else {
     penetrationTestSequence[j] :=
                        functionalTestSequence_t[i];
     j := j + 1;
  }
}
```

The *substituteValues*() function is where the values of the fuzz vector are used to modify the original corresponding functional testing event. In our case the fuzz vector is a modification of the maliciousForm template where the password is replaced by the SQL injection value. The above code is used to set the template content. In the actual code generation step, we use the same matching mechanism against checkpoint events in order to generate the verdicts associated with a checkpoint event as shown in section 3:

5) Generating test component coordination

In [16] we have specified a precise test behavior architecture with two main test components, one for the victim and one for the attacker. Naturally, the events of both components are not randomly interleaved. Some attack vectors require a very specific order in the sequence of events drawn from the two components behavior. The process we have described in the previous section consists mainly in composing a new sequence from two individual sequences of victim and attacker behavior. Thus, we need to apply another procedure first to separate this combined events sequences into separate sequences and have the MTC steer the individual components.

This code generation is relatively trivial as the events data of our combined sequence has a field that specifies which component performs the event. Thus, a simple scan of the combined event sequence is used to write the appropriate event in a separate file. The coordination messages are generated for both the MTC behavior and each component every time the component changes in the combined sequence. For example, in the following combined sequence of events:

```
{ {event_1, "attacker"}, {event_2, "victim"},
       {event_3, "victim"}, {event_4, "attacker"} }
```

This sequence will result in an MTC coordination messages sequence that sends a coordination message to the component that should start performing the event and then receives a confirmation message that the event has indeed been performed as follows:

```
function MTC_behavior() … {
  …
  attackerCoordPort.send("perform event_1");
  attackerCoordPort.receive("event_1 performed");
  victimCoordPort.send("perform event_2");
  victimCoordPort.receive("event_3 performed");
  attackerCoordPort.send("perform event_4");
  attackerCoordPort.receive("event_4 performed");
}
```

These same coordination messages are mirrored in the individual components behavior in order to block the execution of events as long as other components have not performed their behavior fragments.

VI. Conclusion

In this paper, we have demonstrated how to leverage the advantages of TTCN-3 in a model-based approach to web application penetration testing by addressing two types of penetration test cases: SQL Injection and XSS attacks. The TTCN-3 testing language through its separation of concerns approach allows specifying both models and test cases in a highly abstract way. Abstraction is the key factor to reduce modeling efforts and test results analysis. Also, the TTCN-3 engine and run-time environment supports the creation of clear and concise testing architectures that maximize re-usability in a sound and systematic fashion. Finally, it allows a divide-and-conquer approach of the modeling activities by separating the problem into several concurrent models.

References

[1] OWASP TOP 10: The Ten Most Critical Web Application Security Vulnerabilities. Retrieved December 2011 from The Open Web Application Security Project: http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf

[2] TOP 20 Internet Security Problems, Threats and Risks. Retrieved December 2011 from The SANS Institute: http://www.sans.org/top20/

[3] The WASC Threat Classification v2.0. Retrieved December 2011 from Web Application Security Consortium: http://projects.webappsec.org/f/WASC-TC-v2_0.pdf

[4] OWASP Testing Guide. Retrieved December 2011 from The Open Web Application Security Project: https://www.owasp.org/images/8/89/OWASP_Testing_Guide_V3.pdf

[5] Palmer, S. (2007). Web Application Vulnerabilities: Detect, Exploit, Prevent. Syngress Publishing.

[6] 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

[7] Testing Technologies, TTworkbench - an Eclipse based TTCN-3 IDE, www.testingtech.com/products/ttworkbench.php, 2011.

[8] ISO 8807, Language Of Temporal Ordering Specification, 1990

[9] ITU-T Z.100, Specification and Description Language, 1988

[10] OMG, Unified Modeling Language, http://www.omg.org/spec/UML/

[11] G.Din, K-D. Engel, A. Rennoch, An Approach for Test Derivation from System Architecture Models applied to Embedded Systems, in MoTiP proceedings, 2009

[12] Arkin, B., Stender, S., & McGraw, G. (2005, Janunary-February). Software Penetration Testing. IEEE Security & Privacy, Volume 3 (Issue 1), pp. 84-87.

[13] M. Utting, A. Pretschner, B. Legeard, A Taxonomy Of Model-Based Testing, working paper http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf

[14] B. Stepien, L. Peyton, P. Xiong, Framework Testing of Web Applications using TTCN-3, International Journal on Software Tools for Technology Transfer, Springer Berlin / Heidelberg. Vol. 10, No. 4, pp 371-381, 2008

[15] P. Xiong, L. Peyton, "Model-Driven Penetration Test Framework for Web Applications", Eight International Conference on Privacy, Security and Trust, Ottawa, Canada, August 2010.

[16] B. Stepien, P. Xiong, L.Peyton, A Systematic Approach to Web Application Penetration Testing Using TTCN-3 in G. Babin, K. Stanoevska-Slabeva, P. Kropf (Eds.): MCETECH 2011, LNBIP 78, pp. 1–16, Springer-Verlag Berlin Heidelberg 2011

[17] J. Ernits, R. Roo, J. Jacky, M. Veanes, Model Based Testing of Web Applications using NModel, in TESTCOM/FATES 2009 proceedings

[18] Manzuik, S., Gold, A., & Gatford, C. ,Network Security Assessment: From Vulnerability to Patch. Syngress Publishing, 2007

[19] Splaine, S., Testing Web Security: Assessing the Security of Web Sites and Applications. John Wiley & Sons, 2002

[20] Jacky, J., Veanes, M., Campbell, C., & Schulte, W., Model-Based Software Testing and Analysis with C#. Cambridge University Press, 2008

[21] A. Tegne, L. Peyton, Model-Based Engineering of a Managed Process Application Framework in G. Babin, K. Stanoevska-Slabeva, P. Kropf (Eds.): MCETECH 2011, LNBIP 78, pp. 173–188, Springer-Verlag Berlin Heidelberg 2011

[22] Behnam, S.A., Amyot, D., Forster, A.J., Peyton, L., and Shamsaei, A., Goal-Driven Development of a Patient Surveillance Application for Improving Patient Safety, 4th International MCeTech Conference on eTechnologies, Ottawa, Canada, May, 2009. LNBIP 26, Springer, pp 65-76.

[23] Webgoat Project Page, http://code.google.com/p/webgoat/

[24] Jagadish. S., Lawrence, C, Shyamasunder R.K, cmUML - A UML based Framework for Formal Specification of Concurrent, Reactive Systems, Journal of Object Technology (JOT), Vol. 7, No. 8, Novmeber-December 2008, pp 188-207. http://www.jot.fm/issues/issue_2008_11/article7.pdf

[25] Thomas, H., & Chase, S. (2005). The Software Vulnerability Guide. Charles River Media

[26] Bishop, M. (2007, November-December). About Penetration Testing. *IEEE Security & Privacy, Volume 5*(Issue 6), pp. 84-87

[27] Arkin, B., Stender, S., & McGraw, G. (2005, Janunary-February). Software Penetration Testing. *IEEE Security & Privacy, Volume 3*(Issue 1), pp. 84-87.

[28] Arkin, B., Stender, S., & McGraw, G. (2005, Janunary-February). Software Penetration Testing. *IEEE Security & Privacy, Volume 3*(Issue 1), pp. 84-87.