Model-Based Vulnerability Testing of Payment Protocol Implementations

Ghazi Maatoug
INRIA Nancy Grand Est
615, rue du Jardin Botanique
54602 Villers les Nancy
Cedex, France
ghazi.maatoug@inria.fr

Frédéric Dadeau
FEMTO-ST Institute/
INRIA CASSIS
16 route de Gray,
25030 Besançon, France
frederic.dadeau@femto-st.fr

Michael Rusinowitch
INRIA Nancy Grand Est
615, rue du Jardin Botanique
54602 Villers les Nancy
Cedex, France
michael.rusinowitch@inria.fr

ABSTRACT

We investigate an approach to automate model-based vulnerability testing of payment protocols used by e-commerce applications. We aim to improve the efficiency and performance of logical vulnerability testing. The proposed approach is based on a formal specification of the protocol implementation (SUT) and vulnerability attack scenario exploitation for driving the test execution. This approach is illustrated with a use case example bookshop application and one of the most used payment protocols: PayPal Express.

1. INTRODUCTION

The tremendous increase in online transactions has been accompanied by an equal rise in the number and type of attacks against the security of online payment systems. These attacks exploit hidden vulnerabilities in payment protocol implementations within e-commerce applications. In fact, to guarantee secure online transactions, e-commerce applications integrate third-party services. This is done by implementing a specific payment module within the application core using specification of web API provided by Cashier-as-a-Service (CaaS) companies such as PayPal. However, this integration introduces new security challenges due to the complexity of coordinating an application internal state with those of the component services and the web client across the Internet. Moreover, some web application developers are not very well trained with secure programming techniques. As a result, the security of an application is not necessarily a priority of the design goals [3]. Indeed, while validation focuses on absence of runtime errors, and conformance w.r.t. specifications, security aspects are too often left-aside [14]. This is exacerbated by the rush to meet deadlines in the fast-moving e-commerce world. Therefore, penetration testing is mandatory to reveal eventual logical flaws that might be otherwise have been overlooked during the development phase. This kind of tests can also be achieved by companies specialized in security testing, in pentesting (for penetration testing) as instance. These companies monitor the constant discovery of such vulnerabilities, as well as the constant evolution of attack techniques.

However, it is worth to say that detecting vulnerabilities in a security protocol, and, in our case, in the payment process, remains a most difficult task to perform, as it requires a deep knowledge of the (payment) protocol and the way it is implemented within the e-commerce application. We need more effective methods to improve the efficiency of implementation testing tools, as most of the existing approaches resort to random or manual testing [5]. The work presented in this paper investigates a semi-automatic tool that aims to improve the efficiency and accuracy of logic vulnerability testing, by means of formal specification and abstract attack scenarios (inferred or manually designed). Similar efforts have been reported in SPaCIoS project [13, 1] on different classes of protocols. These approach differ from the one presented here, as they rely on a different language (ASLan) which is more concrete, and thus easier to concretize for test execution. Besides the considered mutations are very similar to classical code mutation operators, while the HLPSL mutations we consider more specifically target the security functions of a protocol. E-commerce payment protocols and attacks have also been extensively addressed in [14], however this work did not attempt to automate the detection of vulnerabilities of the execution of vulnerability test cases.

We aim to improve a recently proposed architecture for automatically compiling abstract attack traces to concrete executable tests on protocol implementations [9]. We have implemented a partly-automated penetration testing platform to detect vulnerabilities on some implementations of PayPal Express payment protocol which is complex and widely used in business transactions. We have succeeded to test an attack scenario on a realistic implementation using formal attack trace generated by the CL-AtSe model-checker [12].

Although not reported here, a similar experimentation was conducted with our technique, on an implementation of Magento [10] containing a logical security flaw that consists in the absence of signature of the purchase amount. To exploit this flaw, a malicious attacker can modify the amount to pay, and purchase expensive items by paying a ridiculous price. This logical flaw was discovered independently by the

2nd Workshop on Hot Issues in Security Principles and Trust (HotSpot 2014). Supported by DAST Project of *Fonds national pour la société numérique- Technologie de Sécurité & Résilience des Réseaux* and *FP7 NESSoS project*.

NBS Systems company[1].

Notice that, although this work addresses payment protocols, it can also be applied to other security protocols such as authentication (see [6] for more details).

In the remainder of this paper, we first present, in Section 2, an overview of the methodology. Then, we discuss the specificity of the addressed type of vulnerability on a concrete example, introduced in Section 3. Section 4 describes in details the principle of the proposed approach. We provide the modelling material and the platform architecture in order to show how the test generation tool chain produces abstract attack traces and how they are reproduced on real-word implementation of an e-commerce application (E-Book Shop) implemented with a vulnerable payment module. Section 5 describes studies on the formal analysis of payment protocols. The strengths of our proposed approach is discussed in Section 6. Finally, Section 7 presents a conclusion and the future works.

2. MODEL-BASED VULNERABILITY TEST GENERATION

We give here an overview of our Model-Based Vulnerability Testing (MBVT) approach as a generic solution for logic vulnerabilities testing.

2.1 Principles of the approach

We first describe the principles of the approach before giving information on the different artefacts that it involves, namely the protocol model, formal attack trace and formal attack scenario. The proposed process to perform vulnerability testing, depicted in Figure 1, is composed of the following activities:

- Formal specification: This activity is done by the security test engineer before starting the test process. It consists in formalizing the system under test (SUT) from the existing specification provided by informal requirements. The formal model is expressed using the HLPSL language [2].
- Mutation process: Mutation [7] is a technique that consists in introducing logical faults, in our case into the HLPSL model in order to create vulnerabilities. These mutations simulate implementation choices or actual mistakes that can be made by a programmer. This can be automated for HLPSL using an existing mutant generator named jMuHLPSL [6].
- Model checking: After applying the mutation process, model checking tools are used to verify the protocol, and possibly generate abstract attack trace if the mutant is declared unsafe. The goal of the model checker is specified using LTL formula while defining specific security property. In this approach, we consider the CL-AtSe model-checker a back-end of the AVISPA protocol analysis tool-set, as this tool is able to produce counter-examples as attack traces if the protocol is declared unsafe. However, other back-ends of AVSIPA could be used, such as On-the-Fly Model-Checker (OFMC) or SAT-based Model-Checker (SATMC).
- Adaptation: Before starting an execution of the formal test scenario, a step is required. Indeed, during the modelling activity, all data used by the protocol are modelled at an abstract level. As a consequence, the attack scenario is expressed at this level, and can not be executed as is. This step thus consists in bridging the gap between abstract keywords, used in the abstract trace, and the real API of the SUT. During this step, the security test engineer has to define how modelled data are implemented within the SUT.
- Test execution: it aims to automatically execute abstract scenarios on concrete implementations. This step relates modelled data existing in the formal attack scenario with the real API defined by the test engineer in the previous step. Communication with the SUT happens in real time and in a dynamic way.

If no informal specification of the protocol exists, the first two steps (formal specification and mutation process) can be replaced by a model inference technique. This can be performed more or less automatically using other techniques such as traffic analysis between agents involved in the execution of the protocol. In this context, the conclusions that can be drawn from the test execution are different. Instead of looking for a vulnerability (introduced at the mutation step), this activity will check that the vulnerability actually exists in the implementation.

2.2 Logical flaws in payment methods

Nowadays e-commerce web applications increasingly integrate a trusted third-party component presented as a Cashier-as-a-Service (CaaS) in the payment process. The main purpose is to better guarantee secured payment transactions, as the CaaS can collect the payment of a purchase from the shopper and inform the merchant of the completion of the payment without revealing the shopper's sensitive data such as a credit card number. In the considered case study, the well-known PayPal server is used as an example of CaaS server.

During a checkout process, communications happen between the third entity and the merchant as well as between these two services and the web client controlled by the shopper. This trilateral interaction is meant to coordinate the internal states of the merchant and the CaaS, since each party has only a partial view of the entire transaction. Unfortunately, this third-party integration introduces a complexity in the payment protocol implementation within the e-commerce application which brings new security issues. Indeed, an improper distribution of the protocol functionality between the involved entities leads to logical flaws that can be exploited by a malicious shopper. Actually, an online purchase transaction is always initiated by the client (web browser) and managed by some public API methods implemented in the two sides: the merchant and the CaaS.

A dishonest shopper can make web API calls of methods existing on the e-commerce application with well-chosen arguments and in an arbitrary order so that he can shop products

[1] `http://www.nbs-system.com/wp-content/uploads/Advisory_Magento_Paypal.pdf`

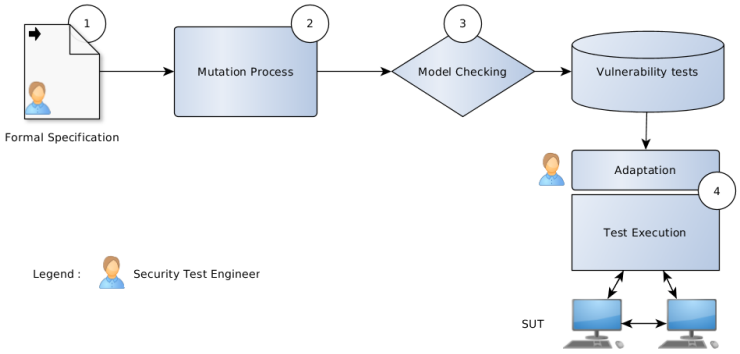

Figure 1: Model-based vulnerability test process

for free or alter the way the payment is verified [14]. This shows that to have communications over https do not prevent sever attacks against e-commerce applications. It is worth to mention that network man-in-the-middle attacks are not considered here, since the checkout modules of all the merchants and CaaS websites communicate exclusively over https.

3. RUNNING EXAMPLE: E-BOOK WITH PAYPAL

3.1 Description of the application

E-Book shop is an e-commerce application that contains a number of vulnerabilities related to logical flaws in payment process. Its main goal is to test the efficiency of proposed approach in a legal environment, simplify the complexity of the testing process and get more hands on the SUT. Hence, we consider this application as a *honeypot*.

E-Book shop has been developed as a real e-commerce application with the following features:

- Authentication: E-book shop provides personalized content to registered users.
- Search: The search feature offers the possibility to filter books by names.
- Purchase Books: A registered user on E-book shop can purchase books using his PayPal account.

E-book shop is an e-commerce application that allows a user to search and select books through a shopping cart system and then purchase the chosen products with his PayPal account. Commands that have been paid successfully are saved in a local data base. Figure 2 provides some screenshots of the application. We implemented the payment module using the PayPal sandbox framework provided by PayPal site, and the integrated vulnerabilities was a subject of a deep research on the last revealed security flaws in the most used e-commerce applications.

We now detail one of the attack scenarios we used during the testing phase of the SUT.

3.2 Example of a concrete attack scenario

The concrete attack scenario we consider enables a dishonest user to purchase an expensive product and pay for a cheaper one. This attack exploit data freshness vulnerability integrated within the payment module implemented in the E-Book Shop application. Figure 3 depicts all the exchanged messages between different entities involved in the payment process. The scenario consists in initiating two parallel sessions with the system under test (the E-book Shop application) with the same account and no matter if its done with the same browser or not. During the first session, the attacker chooses an expensive product and starts the payment process. However, he stops at the login step on the sandbox of PayPal. On the other session, the attacker starts the payment steps for a cheap product. When he gets the confirmation of payment from the PayPal site, he substitutes the token value with the token of the first session, before getting redirected to the merchant site. This way, the merchant (if incorrectly implemented) believes that the attacker has paid for the expensive product and responds with a successful payment. But, in reality, the attacker has only paid the cheaper product of the second session. We describe in details the process of this attack scenario detection and simulation starting from formal modelling ending to the real simulation on a concrete implementation.

4. DETAILS OF OUR MBVT APPROACH

In this section, we detail each main activity of the Model-Based testing process. For each activity, we present its objectives as well as its process. The E-book shop running example is used to illustrate our approach.

4.1 Modelling the SUT

In order to conduct the security analysis of the PayPal payment protocol, the approach starts by specifying the protocol relying on Alice-Bob notation. The checkout process begins when the button "Pay Now" on the merchant web site is clicked. This operation directs the shopper's browser to the PayPal website where he is invited to provide his PayPal buyer account credentials to continue the purchase process. If the information entered by the user is correct, the shopper is again redirected to a payment success merchant Web page. Behind the scene, there are HTTP interactions between the three parties, who communicate by calling Web-APIs

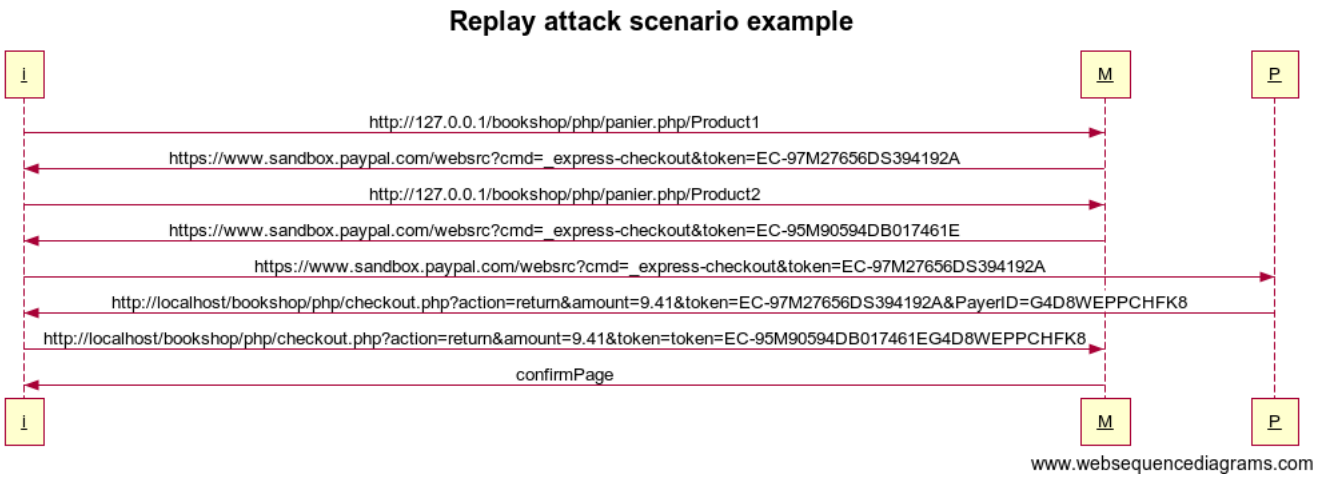

Figure 3: Example of a concrete attack scenario on PayPal Payment implementation

exposed by the merchant and the CaaS. Such APIs are essentially dynamic web pages and are invoked through HTTP requests. A client sends an HTTP request through an URL with a list of arguments and receives an HTTP response, often a Web page dynamically built by the server, as the outcome of the call. The formal model of the protocol was designed using the PayPal documentation [11] and some traffic analysis. To examine the traffic, E-Book Shop application was deployed on a local xampp server [15] and HTTP traffic capturing tools were used, such as Fiddler [8], retrieving all the HTTP exchanged messages between the involved entities during a checkout process. Figure 4 provides the Alice-Bob notation of the PayPal Express protocol, which is described in an associated HLPSL specification (not shown here). For the purpose of efficiency and conciseness, only the most significant steps and fields were modelled. In this figure, C denotes the client (web browser), M is the Merchant (E-book shop) and C is the CaaS (PayPal).

First, the client starts by login to the application, choosing the product and initiate the payment this is modelled using the message checkout. It contains also the order description informations (product details, shipping address, billing address...). The merchant redirects the client to the PayPal site using a `redirectUrl` which corresponds to the `paypalconnect` abstract message in Alice-Bob notation. Also, the latter denotes the login process to PayPal account and the payment confirmation action on the PayPal site. In addition, the merchant generate a token which is a random value used to identify the payment in process. Once the the previous step is validated by the Paypal server, the latter responds with 302 HTTP redirection to the `returnUrl` url specified by the merchant, the `Token` received from the client and the `PayerID` which identify the client PayPal account. Finally, the merchant confirm the success of the payment process. During the execution of the protocol PayPal Express, communication happen also between the merchant and the PayPal site in two times. First, the `SetExpressCheckout` message, sent by the merchant, informs the PayPal server about an upcoming operation. Then, the `DoExpressCheckoutPayment` message requests the payment execution.

4.2 Model checking and test generation

The main purpose of the test generation activity is to produce test cases from the specified model. After modelling the protocol, the model checking tools are used to verify the HLPSL model, and possibly generate abstract attack traces if unsafe. The goal given to the model-checker is specified using HLPSL `witness` and `request` features. These latter inform the CL-AtSe model-checker to ensure that the `Token` value is generated in a fresh manner during the protocol execution, CL-AtSe finds the attack trace that violates the test purpose related to the specified HLPSL model of the SUT.

Figure 5 depicts the attack scenario at the formal level. It corresponds to a replay attack on the token value. Notice that, for this example, the mutation phase explained in Section 2.1 has not been applied. However, such a model could result from a correct model that would have been mutated.

4.3 Adaptation

Before starting an execution of the formal test scenario, a preliminary work step is required. During the modelling activity, all data used by the protocol are modelled at an abstract level. As a consequence, the attack scenario can not be executed as is. The gap between abstract keywords used in the abstract trace and the real API of the SUT must be bridged. Indeed, the security test engineer have to define how modelled data is implemented in the SUT. Also, while receiving responses from the SUT, our tool is in charge of retrieving relevant fields from the received response. In case of sending operation, each abstract message needs to be translated into real message format. Table 1 describes the semantics of the abstract data involved in our example of attack scenario, given in Figure 5. Some abstract messages correspond to operations performed by the tool Attack Simulator, others correspond to fields contained in the HTTP messages.

4.4 Test execution : platform architecture

Table 1: Mapping modelled data to its semantic use for testing

Modeled data	Its implementation
checkout	start(); loginToBookShop() ; chooseProduct()
paypalconnect	loginToPaypal()
returnUrl	http://localhost/bookshop/php/checkout.php?action=return
Token	getToken()
PayerID	getPayerID()

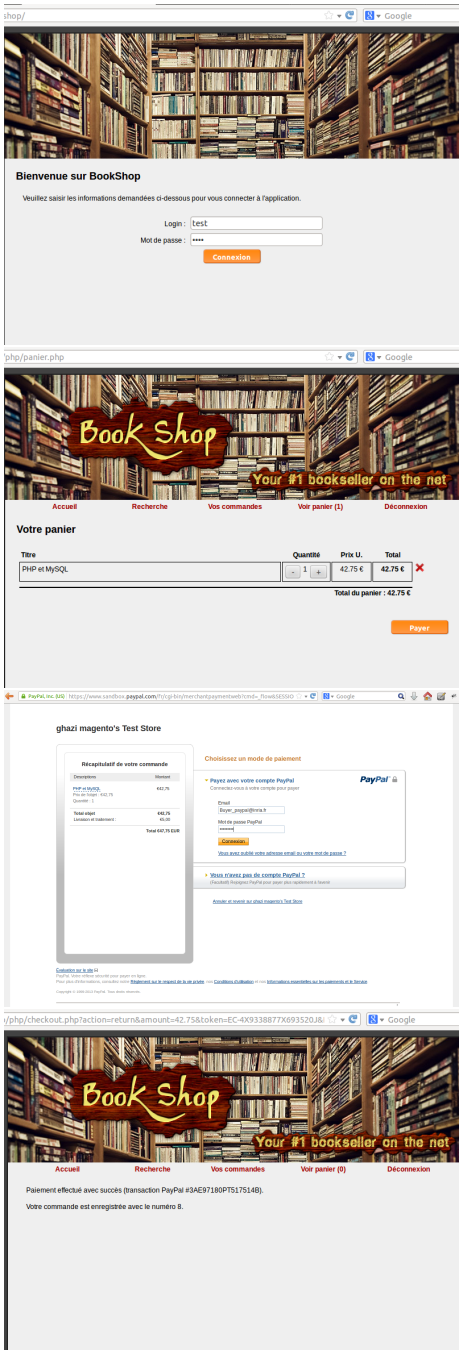

Figure 2: Regular payment process

```
C -> M : checkout
M -> P : SetExpressCheckout
P -> M : Token
M -> C : paypalconnect, Token
C -> P : paypalconnect,Token
P -> C : returnUrl, Token, PayerID
C -> M : returnUrl, Token, PayerID
M -> P : DoExpressCheckoutPayment, Token, PayerID
P -> M : Result
M -> C : confirmPage
```

Figure 4: Alice-Bob notation of PayPal Express protocol

```
i -> M : checkout1
M -> i : paypalconnect, Token1
i -> M : checkout2
M -> i : paypalconnect, Token2
i -> P : paypalconnect,Token1
P -> i : returnUrl, Token1, PayerID
i -> M : returnUrl, Token2, PayerID
M -> i : confirmPage
```

Figure 5: Formal attack trace

As discussed above, the attack trace produced by a model-checker is rather abstract and, in order to be able to detect real attacks that affect protocol implementations, it is mandatory to provide a platform that performs both (*i*) messages format conversion, from a formal level to the implementation level, and (*ii*) real communications with the SUT. This platform's architecture is now described, in terms of components, with their functionalities and interactions. It displays three main components, each with a specific role:

1. *Attack Trace Compiler*: identifies agents, messages types and elementary operations.
2. *Scenario Execution Engine*: generates (resp. retrieves) outgoing (resp. incoming) messages.
3. *Attack Simulator*: simulates the scenario on real communication channels.

As shown in Figure 6, the testing environment takes as inputs the attack trace and the mutated model of the considered protocol, and returns an indication whether the considered attack on the considered implementation exists or not. To better understand the functionalities of each module, we will rely on the previous scenario as a use case example in what follows.

Attack trace compiler

The *Attack Trace Compiler* transforms an abstract attack trace given in Figure 8 into an executable attack scenario described in Figure 9. The component collects intruder initial knowledge, shown in Figure 10, from the HLPSL protocol and follows the attack trace instructions to build the

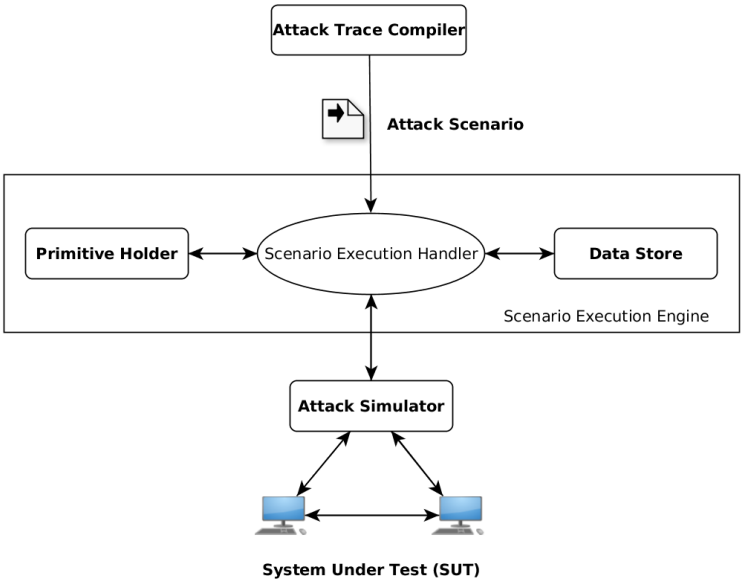

Figure 6: Platform Architecture

```
L1:
7="received at step:1"
8=p1(7)
9=p2(7)
```

Figure 7: Additional information for each attack scenario step

attack scenario. The latter describes in details the actions that should be performed by the intruder when executing the attack. Hence, it is structured into steps and elementary operations, each step corresponding to an abstract attack trace instruction. In order to explain the process of compiling the attack trace, we give here an example of attack trace treatment, based on the detected scenario on PayPal Express payment protocol. Recall that this scenario violates the freshness of the exchanged value of token between the merchant and the PayPal server. For instance, we take the second line in the abstract trace in Figure 8 which corresponds to: `M -> i : paypalconnect, Token1`. It is a receiving operation (denoted by the symbol ?) of the concatenation of the `paypalconnect` and `Token` value. Thus, the output of the attack trace compiler is as follows: $step : 1\ ?7 = pair(paypalconnect, Token\_g7)$. Also, the attack scenario contains additional informations about how to get each part of the composed message described in Figure 7. We denote here p1(m) (resp. p2(m)) the projection of a message m on the first (resp. second) component. In addition, the attack scenario contains a data structure, shown in Figure 10, which is initialized with intruder *Initial Knowledge* (keys or agent's identities) and is updated each time the system evolves to a new state.

Scenario execution engine

This module is responsible of translating the attack scenario, from the formal level to the implementation level. Since the

```
i -> (m,4):  Checkout_g1

(m,4) -> i:  pair(paypalconnect,Token_g7)


i -> (m,4):  Checkout_g1

(m,4) -> i:  pair(paypalconnect,Token_n1)


i -> (p,5):  pair(paypalconnect,Token_g7)

(p,5) -> i:  pair(returnURL,pair(Token_g7,PayerID_n7))


i -> (m,4):  pair(returnURL,pair(Token_n1,PayerID_g2))

(m,4) -> i:  ConfirmPage_n2
```

Figure 8: Abstract attack trace as generated from the model checker

execution environment is designed for an implementation level, the exchanged messages are real network messages. As seen previously, messages in the formal model are specified as first-order terms. Therefore, it is necessary to map these terms to concrete messages and operations. This is the main role of the *Scenario Execution Engine*: it ensures the association between abstract messages and concrete ones, stored in the *Data Store* module. Operation execution is held with the functionality provided by the *Primitive Holder*. Figure 6 describes more this module components.

The attack scenario instructions can be classified into three categories: (1) message construction, (2) message sending, and (3) message receiving. To do this, cryptographic primitives (*crypt*, *pair* and *unpair*) and network primitives (*send* and *receive*) are used.

```
step: 0
!6= Checkout_g1
L1:
6="generated nonce at step:0"

step: 1
?7= pair(paypalconnect,Token_g7)
L1:
7="received at step:1"
8=p1(7)
9=p2(7)

step: 2
!6= Checkout_g1

step: 3
?10= pair(paypalconnect,Token_n1)

L1:
10="received at step:3"
11=p2(10)

step: 4
!7= pair(paypalconnect,Token_g7)

step: 5
?12= pair(returnURL,pair(Token_g7,PayerID_n7))

L1:
12="received at step:5"
13=p1(12)
14=p2(12)
15=p2(14)

step: 6
!16= pair(returnURL,pair(Token_n1,PayerID_g2))

L1:
18="generated nonce at step:6"
17=pair(11,18)
16=pair(13,17)

step: 7
?19= ConfirmPage_n2
```

Figure 9: Abstract attack scenario

```
L0:
0="initially known"
1="initially known"
2="initially known"
3="initially known"
4="initially known"
5="initially known"
6="generated nonce at step:0"
7="received at step:1"
8=p1(7)
9=p2(7)
10="received at step:3"
11=p2(10)
12="received at step:5"
13=p1(12)
14=p2(12)
15=p2(14)
18="generated nonce at step:6"
17=pair(11,18)
16=pair(13,17)
```

Figure 10: Intruder Knowledge

Primitive holder

The needed cryptographic operations are defined in the *Primitive Holder* module. In relation with the specification of the protocol, this component provides a library of operations such as encryption, decryption, nonce generation, signing and concatenation. It is necessary to make sure that the whole scenario can be executed without any errors. To do that, the Primitive Holder provides all the possible operations needed by the protocol implementation. It is worth to say that in the used protocol example (PayPal Express) as implemented in the E-book Shop application, we do not need cryptographic operation implementation. However, this can mandatory when dealing with more complex e-commerce payment protocols.

Data store

Message creation depends on the knowledge acquired in previous step of the scenario, since the tested protocols are stateful. Hence, all the messages handled by the platform are saved in the *Data Store* in their real format and in an indexed way which facilitates data processing. The Data Store also contains all objects required for intermediate computation like encryption keys, data nonces, agent identities and sub-messages.

Scenario execution handler

This is the platform core algorithm which handles the instantiation of abstract operations by concrete executable one. It takes as input the elementary steps of an attack scenario, and processes each instruction in order to identify the next operation to perform as well as its arguments. It interacts with the Primitive Holder module to execute cryptographic operations and with the Data Store module to save or retrieve arguments depending on the attacker behaviour described in the attack scenario. Algorithm 1 describes all the interactions with different modules.

Input: Instruction I
Output: Request to another component
Let I gets instruction value;
Case {I is send(Xi)} **then**
    Get data from the Data Store at position i ;
    Call A−Simulator to send message
Case {I is Xi=receive()} **then**
    Call A-Simulator to get the received message ;
    Store the message on the Data Store at position i
Case{I is Xi=operation(Xy,Xz)} **then**
    Get data from Data Store at positions y and z;
    Call the Primitive Holder to execute the primitive;
    Store the message on the Data Store at position i
Case{I is finish()} **then**
        exit with success

Algorithm 1: Scenario Execution Handler

The first (resp. second) case corresponds to a message sending (resp. receiving) operation over the network. The third case of Algorithm 1 corresponds to the message construction or decomposition. In all cases, the Handler invokes the Data Store and the Primitive Holder modules. Consider, for instance, instruction $X1 = pair(X2, X3)$. First, the Scenario Execution Handler collects the arguments by

requesting them from the Data Store. Then, it calls the **concatenate** method in the Primitive holder to construct the message. Finally, the latter is stored at the result position $X1$ in the Data Store.

Attack simulator

After mapping a formal message to the real format, the *Scenario Execution Handler* processes emission and reception operations. In these cases, it sends a request to the *Attack Simulator* module, which is the interface of the platform with the external environment. At the formal level, the protocol model abstract some fields existing in a real implementation, which need to be restored at the concrete level. The *Attack Simulator* is in charge of the conformance of the exchanged data with the protocol model, meaning that it has to identify the relevant fields and retrieve data from the SUT response (case of receiving operation) and to instantiate the relevant fields in the request message (case of sending operation). In general, the *Attack Simulator* tasks include: (*i*) creating the real communication channels, (*ii*) sending messages, and (*iii*) receiving messages. Therefore, HtmlUnit was integrated in the Attacker Simulator module. HtmlUnit is a Java unit testing framework for testing Web based applications. This headless browser allows Java test code to examine returned pages either as text, as XML DOM, or as collections of forms, tables, and links. It can also deal with HTTPS security, basic HTTP authentication, automatic page redirection and other HTTP headers. Furthermore, this testing framework was used to automate clicks on links and navigation between pages of the online store. We give here an example of HtmlUnit use within Attack Simulator Class. In our case study scenario, step 0 of the abstract attack scenario corresponds to the following informations:

```
step: 0
!6= Checkout_g1
L1:
6="generated nonce at step:0"
```

After generating the message nonce using the information provided by L1 and the operation implementation within the *Primitive Holder* module, the *Attack Simulator* proceeds with the sending operation. However, as we modelled the protocol using alice-bob notation *Checkout* denotes the steps of login to the application, choosing product randomly and starting payment process. Therefore, we provide the sending operation manually implemented in the *Attack Simulator* with the corresponding HtmlUnit fragments of code in Figure 11.

The attack validation

Attack validation is the most important step of the testing process, as it affects the efficiency of the proposed tool in attack detection. The simulator needs to assert whether the attack is simulated with success or not. This must be done in a rigorous way to avoid false positives and false negatives. Mainly, we propose to identify the final state of the SUT when the attack succeeds. Due to the complexity of this process, we have studied two cases with different verdict assignment methods:

1. *First case*: when the attack success is achieved by reaching a final state which is known by the test engineer and can be identified with a simple verification using a verdict keyword. In this case, upon completion of the attack scenario execution, the simulator asserts whether the final state of the system is the state corresponding to an attack success or not. This was done using the JUnit testing framework that helps deploying the attack validation process as follows:

   $String\ verdict;$
   $assertTrue(currentpage.asText().contains(verdict));$

2. *Second case*: the attack success validation is not trivial and one needs to verify all informations about payment operations such as payment status, seller PayPal account situation, etc. Therefore, we propose to provide the test engineer with a log file which contains all the activities performed by the platform while the test execution. This facilitates the task of asserting whether the attack scenario was simulated successfully or not.

```
  public void send(Object object) {

String name = ((NameValuePair) object).getName();
String value = ((NameValuePair) object).getValue();

if (value.equals("Checkout")) {

HtmlPage currentpage = webClient.getPage(baseUrl + "/index.php");

HtmlTextInput textinput = (HtmlTextInput) currentpage
.getElementByName("login");
textinput.setValueAttribute("test");

HtmlPasswordInput passwordinput = currentpage
.getElementByName("password");
passwordinput.setValueAttribute("test");

HtmlSubmitInput submit = (HtmlSubmitInput) currentpage
.getElementByName("btnConnexion");

currentpage = submit.click();

List<DomElement> products = currentpage
.getElementsByIdAndOrName("btnAjouterPanier");
int max = products.size();
Random randomGenerator = new Random();
int i = randomGenerator.nextInt(max);

HtmlSubmitInput submit = (HtmlSubmitInput) products.get(i);

currentpage = submit.click();
HtmlSubmitInput submit = currentpage.getElementByName("btnPayer");
currentpage = submit.click();
}
```

Figure 11: Java code implementing the sending operation

5. FORMAL ANALYSIS: AN EXAMPLE

As specified in the first section of this paper, the testing process using our approach starts with formal specification of the SUT. Then we proceed with the model checking verification and validation step in order to generate abstract attack trace. The latter serves as in input for our testing platform.

In this section, we present the formal analysis work done on the most used payment methods (Paypal Payment, Amazon Payment, Google Checkout). Mainly we focus on how for-

mal verification and validation techniques can help finding implementation logical flaws. In our work, we rely on a list of recently discovered implementation logic flaws provided by [14]. Specifically, we study an other example of attack scenarios on payment module: *Integration of Amazon Simple Pay : paying to the attacker himself to check out from the victim*

Amazon Simple Pay payment protocol is one of the leading payment methods implemented in merchant web sites. Figure 12 shows the workflow while executing the checkout process.
After choosing a product the shopper starts payment process by clicking on the pay button. Then the merchant redirects the shopper's browser to the payment API of the CaaS, passing `orderID`, `gross` and `returnURL` as the arguments. This message is signed by the merchant, so the shopper cannot tamper with the arguments when forwarding the message to Amazon site. After the CaaS (i.e., Amazon) verifies the merchant's signature, the shopper makes the payment, which the CaaS records to its database. The payee is the merchant who signs the merchant redirection message. Then, the CaaS redirects the shopper back to the merchant using the `returnURL` that the merchant supplies in the redirection message. The entire CaaS redirection message is signed by the CaaS, which is verified by the merchant. This checkout procedure seems secure as no data can be contaminated by the attacker.

Flaw and exploit. In fact, this protocol implementation can be vulnerable when the malicious shopper also plays the role of a different merchant. Specifically, anyone can open a seller account on Amazon. Suppose that the seller account is registered under the name "Alice". What the attacker wants to do is to pay Alice (actually, himself) but check out an order from a store belonging to Bob (https://Bob.com). The attack proceeds as follows. First, the attacker starts a payment process and blocks the redirection message provided by Bob. Then, he decrypts the received message, and acting as "Alice", he signs it with his private key. The trick here is that the message signed by Alice actually carries a `returnURL` to Bob (Bob.com/finishOrder). As a result, even though Alice (the attacker) is the party that receives the payment, the CaaS will redirect the shopper's browser to Bob with a redirection to call `finishOrder`. Although the message is indeed sent to Bob, it is actually about the payment that the attacker made to Alice. The logics in `finishOrder`, as sketched in Figure 12, does not verify that the payment was made to Bob, and therefore is convinced that the order has been paid. Fundamentally, the problem comes from the confusion between the merchant and the CaaS about what has been done by the other party.

Goal specification using LTL formula in HLPSL. After the modelling step in HLPSL, we proceed with the model-checking step. We use CL-AtSe tool to detect the attack trace that exploits the exposed vulnerability. This is done using a logic formula to invalidate as a goal definition. In fact, goals in HLPSL language serves as test purposes for CL-AtSe tool. The latter executes the specified protocol and tries to find an attacker behavior that violates the property defined in the goal declaration section of HLPSL specification. In the studied example, we define the security property as follows:

```
goal
[] ( (pay(a,Km1,amazon_merchant,RedirectLink) /\
iknows(inv(Km1)) /\
deliver(km,a,amazon_merchant,RedirectLink)) /\
~ equal(Km1, km)
      => pay(a,km,amazon_merchant,RedirectLink)
)
end goal
```

we denote by `km` (resp. `km1`) the public key of a merchant `m` (resp. `m1`). Also, "a" represents the identity of the agent playing the role of the payment service provider (Amazon in our case). In addition, to define the goal in HLPSL model we use two meaningful keywords: pay and deliver. Typically, pay(a, km1, protocol id, text) is the HLPSL fact stating that the agent a has paid the agent whose public key is Km1. In other words, agent a has paid the agent m1. Also, deliver(Km, a, protocol id, text) is the fact stating the purchase delivery by the agent whose public key is Km to the agent a. In order to explain the specified goal we give the following attack state section the IF file [4] .

```
section attack_states:

attack_state ltl_1_1 (Km1,RedirectLink) :=
 pay(a,Km1,amazon_merchant,RedirectLink).
 iknows(inv(Km1)).
 deliver(km,p,amazon_merchant,RedirectLink) &
 not(idequal(Km1,km)) &
 not(pay(a,km,amazon_merchant,RedirectLink))
```

As a result, the attack state corresponding to the specified security is a state when an agent m1 controlled by the intruder receives a purchase payment although he did not deliver any product. Besides, a honest merchant m makes a purchase delivery and did not receive any payment action.

6. DISCUSSION

While the detection of vulnerabilities can be discharged by protocol analysis tools, such as AVISPA, performing vulnerability tests using penetration test tools remains the most difficult task when trying to ensure security of protocols implementations. Such a process is usually tedious and time-consuming, requiring advanced knowledge in software debugging and reverse engineering. There are many cases where no access to the source code/binaries is possible, and where a "black box" kind of testing is the only viable solution. The best penetration testers intimately understand each and every attack used by their automated testing tools; they intuitively and explicitly know what to look for when assessing the results of their tests; they understand how complex software systems work.

The proposed approach for vulnerability testing protocol implementations is oriented towards the following objectives:

1. *Stateful testing*: our tool performs testing on real word implementation in a dynamic way. Also, it is able to

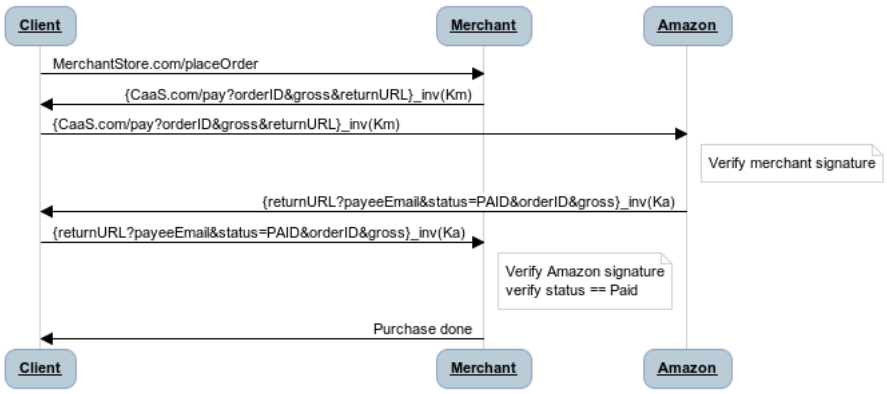

Figure 12: Amazon Pay workflow

simulate multiple session connexions with the SUT unlike most of the state of the art automatic penetration tools.

2. *Accuracy and precision*: our approach is guided by a preliminary formal analysis and model-checking step. Exploitation of powerful formal verification techniques leads to pertinent testing process. The tool helps not only in detecting known implementation logic flaws but also in discovering new ones.

3. *Time efficiency*: the automated tasks in the approach reduces test execution time compared to alternative penetration testing tools. Approximatively two days of engineering work were needed to specify a protocol model and define a security goal. Then model cheking can be performed in a few seconds. Two days of software engineering work were required to develop an adaptation layer.

4. *Scalability*: the tool architecture can easily be extended to cover other protocol implementations. In fact, when dealing with multiple implementations of a specified protocol, one need to adapt the modelled data with its semantics within the SUT. In such way, the database of formal attack traces (inferred or manually generated) is reusable for all implementations corresponding to the specified protocol. See Figure 13.

5. *Discovering implementations*: the construction of a reusable database of attack scenarios helps to perform too some reverse-engineering on protocol implementations. The traces in the database correspond to different specifications (mutated models) and different implementations. The black box testing process helps to derive information about the SUT implementation depending on the executability of the replayed attack traces. For instance we can distinguish between Paypal Standard and Paypal Standard with IPN by the ability to execute or not some scenario from the database.

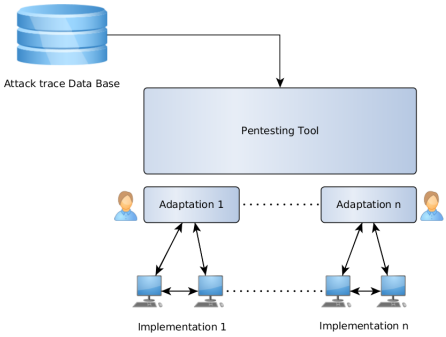

Figure 13: Testing different protocol implementations

7. CONCLUSION AND FUTURE WORKS

In this paper, we proposed an approach that supports the binding of specifications of payment protocols to actually deployed implementations through formal model compiling and the automatic penetration testing of real implementation against putative attacks found by model checker. The approach consists in model checking a (possibly) mutated formal model, looking for attack trace violating security property. If an attack is returned, the platform generates automatically concrete attack scenario instructions, encoding how to verify and generate protocol messages. The abstract attack trace is analysed and the instructions are executed accordingly. In order to assess the effectiveness of the proposed platform, we implemented its architecture relying on the RUP Agile process. We used the Java languages and libraries to implement its components. Especially, to provide the needed functionalities and operations we used the HtmlUnit library. Our platform is able to successfully execute an attack on a PayPal Express implementation within a realistic e-commerce application E-Book Shop. In particular, we applied a replay attack scenario that was detected at a formal level using model checking. It is worth to say that our solution is not specific to a single scenario, and it is able to simulate all the possible formal attack traces related to the modelled PayPal Express protocol. Following our work, we are planning several improvements. In order to simplify the platform use, we will develop a graphical user interface. Also, we will perform further penetration tests on other payment protocol implementations for revealing undiscovered security flaws. To do so, we first need to generate further formal attack traces related to different security properties. We can refer to the mutation techniques

described in [6], applied to a formal specification of the protocol. To achieve that, the model needs to be developed without any knowledge of a concrete implementation (only based on requirements documents). The mutations will inject concrete faults that could represent implementation errors, at the model level, that the test will search for, at the implementation level. Second, we will need to manually adapt the modelled data to its semantic use in the protocol implementation.

8. REFERENCES

[1] A. Armando, G. Pellegrino, R. Carbone, A. Merlo, and D. Balzarotti. From Model-Checking to Automated Testing of Security Protocols: Bridging the Gap. In A. D. Brucker and Jacques Julliand, editors, *Tests and Proofs*, volume 7305 of *Lecture Notes in Computer Science*, pages 3–18. Springer Berlin Heidelberg, 2012.

[2] AVISPA project, Delivrable 2.1. *The High Level Protocol Specification Language*, 2003. `http://www.avispa-project.org/delivs/2.1/d2-1.pdf`.

[3] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.

[4] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF Toolset. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 131–132. Springer Berlin / Heidelberg, 2004.

[5] M. Büchler, J. Oudinet, and A. Pretschner. Semi-Automatic Security Testing of Web Applications from a Secure Model. In *Sixth International Conference on Software Security and Reliability (SERE 2012)*, pages 253–262. IEEE, 2012.

[6] F. Dadeau, P.-C. Héam, and R. Kheddam. Mutation-based test generation from security protocols in HLPSL. In *ICST'11*, pages 240–248. IEEE Computer Society, 2011.

[7] Richard A. DeMillo. Test adequacy and program mutation. In *ICSE*, pages 355–356, 1989.

[8] Fiddler: The free web debugging proxy for any browser, system or platform. `http://fiddler2.com/`.

[9] Hatem Ghabri, Ghazi Maatoug, and Michaël Rusinowitch. Compiling symbolic attacks to protocol implementation tests. In *SCSS*, pages 39–49, 2013.

[10] Magento community edition. `http://www.magentocommerce.com/`.

[11] Paypal development & integration guides. `https://developer.paypal.com/webapps/developer/docs/classic/products/`.

[12] Mathieu Turuani. The CL-Atse Protocol Analyser. In *Term Rewriting and Applications - Proc. of RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 277–286, Seattle, WA, USA, 2006.

[13] Luca Viganò. The SPaCIos project: Secure provision and consumption in the internet of services. In *IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013*, pages 497–498, 2013.

[14] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 465–480, Washington, DC, USA, 2011. IEEE Computer Society.

[15] XAMPP an easy to install Apache distribution. `http://www.apachefriends.org/en/xampp.html`.