2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

VERA: A flexible model-based vulnerability testing tool

Abian Blome, Martín Ochoa
Siemens AG, Germany
firstname.lastname@siemens.com

Keqin Li
SAP, France
Keqin.Li@sap.com

Michele Peroli
University of Verona, Italy
michele.peroli@univr.it

Mohammad Torabi Dashti
ETH, Switzerland
torabidm@inf.ethz.ch

***Abstract*—There exist an abundant number of tools for aiding developers and penetration testers to spot common software security vulnerabilities. However, testers are often confronted with situations where existing tools are of little help because a) they do not account for a particular configuration of the SUT and b) they do not include tests for certain vulnerabilities. To cope with this we propose a tool that allows users to define attacker models where the payloads and the behavior are cleanly separated and that abstract away from low-level implementation details such as HTTP requests.**

***Keywords*-Security; Testing; Extended finite state machines**

I. Introduction

Usually, in order to state what is meant by the "security" of a system, one defines high-level security requirements, determined by the assets that one wants to protect. The goals are integrity, confidentiality and availability of the resources associated with those assets. The requirements are enforced via mechanisms that provide authentication, authorization, secure channels, etc. As a consequence, secondary goals appear, for instance "only authorized people have access to a certain interface, a directory, or to a certain cookie". Goals break down into mechanisms plus new more lower-level requirements. At the end one may have a goal of the type: "the system must not have cross-site scripting (XSS) vulnerabilities" (shorthand for: the system should not react in a particular way to a particular input), because violating this goal might result in a violation of the higher level security requirements (such as confidentiality or integrity breaches).

In many cases it is not always necessary to find a full-blown attack, but it is enough to find low-level vulnerabilities. This is among others justified by the cost/benefit relation of vulnerability testing: usually the presence of a vulnerability is a pre-requisite to deploy an attack, but to actually exploit a vulnerability is a time-consuming task (if at all possible). Consider the following example: A tester finds that when giving some inputs to a web application, a malformed SQL statement error is reported. This could mean that there is a way to craft the input with malicious SQL Injections, but to find the exact form in which these injections are successful is in general not trivial, in particular if the source code is unknown (as in black-box testing). Preventively fixing the application to eliminate the unwanted behaviour (SQL error) offers a reasonable compromise between security and the testing effort.

Thus, security testing often concentrates on vulnerability testing. As of today, there exist an abundant number of tools for aiding developers and penetration testers to spot common software security vulnerabilities. However, testers are often confronted with situations where existing tools are of little help because a) they do not account for a particular configuration of the SUT or b) they do not include tests for certain vulnerabilities. For instance, the SUT could use a particular authentication mechanism (such as a proprietary protocol) and a recent/rare database version. It is likely that most available tools will not cover the authentication mechanism and might not have information about known vulnerabilities of the database model used by the SUT.

In this situations the tester would benefit from extension mechanisms to the available tools. Some tools (like the non-free version of Burp [1] and w3af [2]) allow to write such extension plug-ins. These plug-ins must be written in the programming language of the tool and imply the learning curve of the tool's API. The alternative consists in directly writing scripts for the task in hand.

In this paper we propose a tool named VERA, standing for "VERA Executes the Right Attacks", which allows testers to define attacker models by means of extended finite state machines (EFSM). In this way, testers can define new tests where the payloads and the behavior are cleanly separated and that abstract away from low-level implementation details such as HTTP requests.

II. Modeling

In the following we give a formal syntax and semantics for attacker models that can be executed using VERA, as originally described in [3]. Here the goal is to illustrate the precise meaning of VERA models and not to perform formal reasoning on them. Attacker models can be seen as an extension of Mealy machines [4] with guarded transitions and variables. Similarly, one can see similarities between the attacker models and UML statecharts. The formal syntax and semantics given below however clarifies the differences between attacker models and Mealy machines or UML statecharts (there are many different semantics for UML statecharts; e.g. see [5], [6]). In particular, attacker models do not have a notion of composition.

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.65

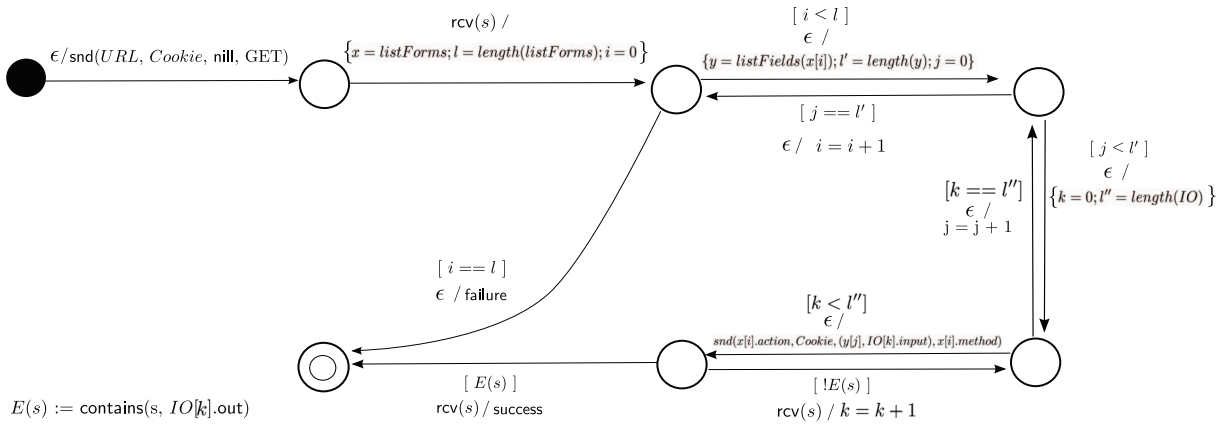

Figure 1. General injection attacker model

A. Example

To begin, we illustrate the semantics by using the example in Fig. 1. In the first transition the attacker starts the interaction with the system by sending a message requesting a particular URL. Subsequently, a message is received by the attacker that does not trigger an immediate reaction, but only a change in the configuration. With the new configuration (the new values of x, l and i) we have two possible transitions: i) to the final state and ii) the to the third state.

The transition to the final state is triggered when there are no more fields to check (or the page pointed by URL doesn't contain any fields), and the attacker gives up by outputting failure. The two transitions, outgoing and ingoing, to, and from represent a configuration change for the variables j, l' (for the outgoing vertex), and i (for the ingoing vertex).

In the lower-right state the attacker sends a message with the j-th payload in IO (to be delivered to the i-th field in the page) without receiving a message.

Subsequently the attacker receives a message and changes his state (thus he reacts) according to its content. If the message doesn't contains the expected output the state will change in order to check the next payload or the next field; on the other hand if the attacker receives the expected output he has found a vulnerability thus outputting success.

B. Formalization

Fix a *term algebra* Σ, where $true \in \Sigma^0$ and $\epsilon, snd, rcv \notin \Sigma$. We write $T_{\Sigma(V)}$ for the *set of terms* induced by Σ using a *finite set of variables* V that is disjoint from Σ. A *substitution* σ is a total function that maps V to $T_{\Sigma(\emptyset)}$. We now introduce guards and commands.

- A *guard* g is a term in $T_{\Sigma(V)}$.
  We assume access to an oracle Ω: Ω is a computable function where for any guard g and substitution σ, we have $\Omega(g\sigma) \in \{0, 1\}$. For any oracle Ω, we require that $\Omega(true) = 1$.
- A *command* c is a term in $T_{\Sigma(V)}$.
  We assume access to a transformation function θ: θ is a computable function where for any command c and substitution σ, $\theta(c, \sigma)$ is a substitution.

An *input event* is either ϵ or of the form $rcv(m)$ with $m \in T_{\Sigma(V)}$. An *output event* is either a finite (possibly empty) list o of commands, or of the form $snd(m)$ with $m \in T_{\Sigma(V)}$. As a notational convention, we write ϵ for the empty list of commands.

An *attacker model* is a finite directed graph whose edges are all annotated with triplets of the form $[g]i/o$ where g is a guard, i is an input event, and o is an output event. We write i/o as a shorthand for $[true]i/o$. Moreover, one or more nodes of the graph are designated as *initial* states. A node with no outgoing edges is called *final*. States are graphically represented as circles. Initial states are denoted by filled circles.

We give an operational semantics for attacker models. Given an attacker model, its configuration is uniquely determined by a node that denotes the current state of the attacker, and a substitution σ that records the values assigned to the variables.

Let s_1 and s_2 be two nodes in an attacker model, where s_1 is connected to s_2 with an edge annotated with $[g]i/o$. The attacker model "moves" from configuration $\langle s_1, \sigma_1 \rangle$ to configuration $\langle s_2, \sigma_2 \rangle$, denoted $\langle s_1, \sigma_1 \rangle \overset{x/y}{\Rightarrow} \langle s_2, \sigma_2 \rangle$, according to the four rules given below.

- Rule 1 :

$$\frac{s_1 \overset{[g]\epsilon/o}{\rightarrow} s_2 \quad \Omega(g\sigma_1) = 1}{\langle s_1, \sigma_1 \rangle \overset{\epsilon/\epsilon}{\Rightarrow} \langle s_2, \theta(o, \sigma_1) \rangle}$$

  where the list of commands in o are composed sequentially as in an imperative programming language. That is, if $o = o_1, \ldots, o_n$ then:

$$\theta(o, \sigma_1) = o_n(\ldots o_1(\sigma_1)\ldots)$$

- Rule 2:

$$\frac{s_1 \overset{[g]\epsilon/snd(m)}{\rightarrow} s_2 \quad \Omega(g\sigma_1) = 1}{\langle s_1, \sigma_1 \rangle \overset{\epsilon/snd(m\sigma_1)}{\Rightarrow} \langle s_2, \sigma_1 \rangle}$$

- Rule 3:

$$\frac{s_1 \overset{[g]rcv(m)/o}{\rightarrow} s_2 \quad \Omega(g\hat{\sigma}) = 1}{\langle s_1, \sigma_1 \rangle \overset{rcv(m\hat{\sigma})/\epsilon}{\Rightarrow} \langle s_2, \theta(o, \hat{\sigma}) \rangle}$$

  where $\hat{\sigma}$ is any substitution such that $\hat{\sigma}(v) = \sigma_1(v)$ for any variable $v \in V$ that does not appear in m.
- Rule 4:

$$\frac{s_1 \overset{[g]rcv(m)/snd(m')}{\rightarrow} s_2 \quad \Omega(g\hat{\sigma}) = 1}{\langle s_1, \sigma_1 \rangle \overset{rcv(m\hat{\sigma})/snd(m'\hat{\sigma})}{\Rightarrow} \langle s_2, \hat{\sigma} \rangle}$$

  where $\hat{\sigma}$ is any substitution such that $\hat{\sigma}(v) = \sigma_1(v)$ for any variable $v \in V$ that does not appear in m.

Intuitively, configurations change according to the attacker's reactions to system outputs. Note that system outputs are inputs of attacker models, and attacker models' outputs are input to the system under test.

An *execution* of the attacker model is a sequence $s_1, x_1/y_1, s_2, x_2/y_2, s_3, \ldots$, where s_1 is an initial state, and $\langle s_i, \sigma_i \rangle \overset{x_i/y_i}{\Rightarrow} \langle s_{i+1}, \sigma_{i+1} \rangle$ for all $i \geq 1$, with $\sigma_1, \sigma_2, \sigma_3, \ldots$ being arbitrary substitutions. We refer to $x_1/y_1, x_2/y_2, \ldots$ as the *input/output* sequence *induced* by the execution.

The semantics of an attacker model is then the set of all input/output sequences actions that are induced by some execution of the attacker model. Note that such input/output sequences are all *ground*, i.e. contain no variables.

Note that a transition annotated with $[g]rcv(m)/o$ may apply only if the received message can instantiate the variables in m (if any), and the guard g is satisfied *after* the instantiation. Otherwise, the received message will be ignored, i.e., the attacker model consumes the received message without making any transitions. This completes the attacker model with respect to incoming system messages, in case some messages are not explicitly covered. A similar remark applies to $[g]rcv(m)/snd(m')$.

We remark that the purpose of our formalization of the syntax and semantics of attacker models is to give a basis to correctly interpret and execute them. Currently we do not intend to perform any formal reasoning on attacker models. For that purpose, it is then enough to consider non-concurrent models representing the reactions of an attacker to a system's outputs. Indeed, we consider only single monolithic attacks without a notion of composition.

III. Implementation

The VERA tool is an extensible framework based on the concept of extended finite state machines that allows the creation and execution of attacker models targeting generic vulnerabilities of web applications in a black-box fashion, in essence reproducing a penetration testers actions. These attacker models can be collected in libraries targeting specific vulnerability types across multiple types of web applications. Besides allowing semi-automatic online testing using the provided libraries, VERA can also be used in fully automated scripts by interfacing directly with the provided back-end or importing and using the libraries provided by the framework.

The back-end of the VERA framework consists of a python program which parses:

1) Selected **Instantiation library** containing data values used to interact with SUT.
2) **Configuration file** containing system specific information needed to test a SUT.
3) **Model file** An XML file containing the attacker model to be tested.

Then the framework creates an instance of the attacker model, and runs it online against the SUT by using the values from the three files and the user if needed.

A. Instantiation library

While the basic functionality of an attack is encapsulated within the attacker models described above, an actual attack quite often needs additional data that is ill suited to automata. An example for this would be a list of passwords in a brute force attack. A decision was taken to outsource this type of information, which is not application specific, into standardized instantiation libraries. These can be accessed by the attacker model as arrays, basically instantiating the attacks with specific values.

If an attack, or parts of an attack, can be performed multiple times with different values, in order to test whether one of these values triggers a vulnerability, these values should be moved into an instantiation file. This file can then be extended by the different security experts.

In some cases it makes sense to use multiple instantiation libraries, which can be used by the security expert in different circumstances. We have identified two common scenarios where the use of multiple instantiation libraries helps the security expert during a vulnerability assessment:

- If the same steps can be performed to create completely different attacks, then it makes sense to use different instantiation libraries for these kinds of attacks. An example for this would be the use of injection attacks (e.g. SQL-, X-Path-, Command-Injection). It makes sense to not have a single big library, but rather have different instantiation libraries.
- If there are a large number of values, and these can be divided based on information which might be available during the test time, such as the back-ends used, it might make sense to split the instantiation library. An example for this would be file enumeration, where dif-

ferent instantiation libraries might exist for the different kinds of platforms available.

The VERA framework allows a user to run an instance of the attacker model with an instantiation library, though the GUI allows the user to select a number of instantiation libraries which are then run sequentially. This instantiation library consists of a simple text file containing an array called `IO` of either single values or tuples. Which kind of data is in the array depends largely on the attacker model it was created for. An example instantiation library for file enumeration can be seen in Figure 2.

```
IO=[
  ".htaccess",
  ".htaccess.bak",
  ".htpasswd",
  ".meta",
  ".web",
  "conf",
  "apache/logs/access.log ",
  "apache/logs/access_log",
  "apache/logs/error.log ",
  "apache/logs/error_log",
```

Figure 2. Fragment of a file enumeration instantiation library.

B. Configuration values

While the goal of the attacker models introduced is to be as generic as possible, once the testing has to be performed, the use of system specific knowledge is unavoidable. This information is stored in special configuration files which specify a number of variables and their values in the following format: **Name=Value**. Comments can be added using the # sign. These have to be assigned by the security expert before the tests can be performed.

For convenience, we have defined a number of parameters that should be used by all models, allowing the expert to create a single file containing all necessary values and then run all selected attacker models and their instantiation libraries using that same file.

- **URL** This contains the target URL for the attacker model. Depending on the attacker model, this URL might just be used as a starting point used to crawl through the entire site.
- **Cookie** This contains the cookies necessary for the web application, such as a session ID.
- **Header** If additional headers are needed for the correct functioning of the web application, such as authentication headers, they can be defined in this variable.
- **Domain** The argument for this is a URL restricting the scope of the instantiation library. Only sites within the scope should be targeted by the attacker model.

Creators of new attacker models are strongly encouraged to use these default parameters, and refrain from extending the list unnecessarily, as this would complicate the interoperability of the different attack models used. If additional information is required the user should be prompted.

An example configuration file targeting the SQL-injection lesson from WebGoat can be seen in Figure 3

```
# WebGoat SQL injection lesson
URL="http://localhost:8080/webgoat/attack?Screen=65&menu=900"

# Session ID
Cookie="JSESSIONID=CF9662702E3222185A55871A63F423D7"

# Header containing the auth. data (guest:guest)
Header={"Basic": "Z3V1c3Q6Z3V1c3Q="}

# Scope
Domain="http://localhost/webgoat/"
```

Figure 3. Configuration file for the WebGoat SQL injection lesson.

C. Attacker model

The attacker model has been implemented following the data structure presented in section II. The model is saved as an XML that uses the following tags:

```
<statechart> </statechart>
```

delimiting the parsed attacker model,

```
<node id="start"/>
```

a node in the diagram with its identifier ,

```
<transition from="start"
            guard="True"
            input=""
            output=""
            to="send"> </transition>
```

the transition between two nodes with the corresponding values,

```
<action value="files=[]"/>
```

the actions to perform during a transition.

In the graphical editor the information needed for the correct execution are saved in a SCM file (State Chart Model file).

D. Interfaces

Currently there are two possible ways to use the VERA framework. On the one hand, it is possible to directly call the command line backend written in python. This allows experienced users to use the VERA framework in various ways, for example by integrating it in more complex scripts, parsing the output using command line utilities or creating automated security scans at certain times. On the other hand, a graphical user interface was created which uses Eclipse as a back end. This graphical interface guides the user through the different options and provides the security analyst with an overview of the attacker models as well as the results.

E. VERA-tool: user workflow

The VERA-tool graphical interface follows the user workflow presented in figure 4.

The first decision a user has to make is if he wants to start from scratch with a new model (②) or work on a previously created model (①). In order to successfully create

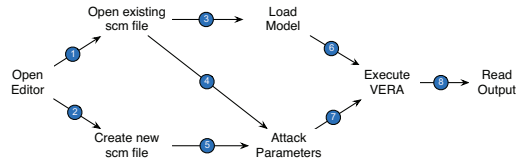

Figure 4. VERA-Tool: user workflow

a new model, the user must use the graphical editor (see section IV-C for an example of the graphical interface used with WebGoat). If the information about the configuration library and the other parameters are missing in the existing model (or in the new model) the user must introduce these information into the editor (④ and ⑤). If the existing model already has the needed information the user must load them (③). After these preliminary steps the tool is ready for the execution (⑥ and ⑦), and the user can retrieve the output from the generated file(s) (⑧).

We can identify in the workflow three different phases in which the user interact with the tool through the following interfaces:

- Graphical: the user must interact with the graphical editor in ① and ②
- VERA-tool menu: the functionalities needed in ③, . . . , ⑦ can be accessed in the menu.
- Graphical/file system: In ⑧ the user can decide how to review the output. The generated file(s) can be opened with a text editor in the file system or via the editor in eclipse.

F. Attack library

In order to efficiently perform tests against systems under validation, it is necessary to have a pre-made library containing a number of vulnerability tests. These can be selected according to the scope of the test. For this reason, we have created a number of different models, some of which are discussed in IV. In the following sections we will give a short overview of the currently available models, and then move on to describe a number of models in further detail.

Code Injection Flaws in HTML Forms: This attacker model describes a general procedure for code injection in *GET* and *POST* parameters of web pages. It is able to parse a web page, select all relevant forms and their input fields, and test these fields for different kinds of reflected injection attacks. By using different types of instantiation libraries it is possible to perform automatic tests for injection attacks such as cross site scripting, SQL injection, X-Path injection, command line injection, LDAP injection and many more.

SQL Injection Attacks: This attacker model is a specialization of the attack *Code Injection Flaws in HTML Forms* attack which specifically target SQL injections. Instead of just checking whether an injection is possible, it tries to extract information from the database back end in order to provide the analyst with additional information about the database.

Source Disclosure: This attacker models tries to trick the web server into not parsing the server side scripts of a web page, and provides the user with a specially crafted URL which contains readable server side scripts.

File enumeration: This attacker model can be used by a security analyst for information gathering: It tries to find configuration files, backups, administrative interfaces and other hidden files and functionalities, and presents the analyst with a list of their URLs.

Remote File Inclusion: This attacker model tries to trick a web server into loading an external file located on the security analyst's host. By monitoring the connections, VERA can tell the security expert whether such an attack was successful.

Cross Site Request Forgery: This attacker model tries to identify CSRF protection tokens for sensitive actions.

IV. Examples

A. Injection vulnerabilities

In this section, we describe a general approach to exploiting different kinds of injection flaws, and provide attacker model that is able to perform a wide range of injection attacks against web applications.

Injection flaws have become one of the most critical type of vulnerability in web applications (number one and two risks in Application security OWASP Top 10 2010[1]). While injection flaws in web application come in various forms, Cross Site Scripting and SQL Injection are among the most famous examples.

The basic principle for exploitability shares a common form in many cases: The input interface contains one or more parameters which are not properly filtered and these parameters get executed by some back-end or, in the case of reflected cross site scripting, by the client itself in the returned page. By sending commands appropriate to the targeted execution engine and analyzing the returned content it is possible to discover indicators of the presence of vulnerabilities, though the quality of indicators greatly depends not only on the output analysis, but also on the input used. It is also important to note, that this technique covers only those kinds of injection attacks that leave an indication of the flaw in the returned content after the vulnerability was exploited. Therefore attacks such as stored cross site scripting or blind SQL injections are, in general, not covered by this attacker model.

Approach: The common mechanism for the type of code injection described above allows a general attack process which takes advantage of the present flaw, irregardless of the targeted execution engine. It influences only the commands being sent as part of the exploitation, as well as

[1]https://www.owasp.org/index.php/Top_10_2010-Main

Input	Output
’42	Unexpected end of command in statement
"42	Unexpected end of command in statement
“42	Unexpected end of command in statement

Table I
SAMPLE INSTANTIATION LIBRARY FOR SQL INJECTIONS

the indicators for the flaw present in the returned content. In the context of the attacker model presented here, the commands being sent can be abstracted as input strings in the instantiation libraries, the indicators as output strings In the case of this model, these are provided by an external instantiation library and worked through iteratively. This allows the use of the model for a wide range of attacks by choosing the right instantiation library based on the expert's knowledge of the SUT.

The attacker model here presented allows a methodological approach to test the presence of injection vulnerabilities of the aforementioned type in the input forms located in the page at that URL. It is worth noting that this approach limits the use of attacker models to web pages containing static forms, excluding approaches such as webservices using json.

Instantiation library: The instantiation library contains a set of tuples consisting of *input* and *output*. The *input* contains values inserted into the target fields containing some kind of injectable code. The *output* contains strings which can be used in order to verify that the injection was successful. Some sample entries can be found in table I.

Configuration values: The model presented below uses the following values from a configuration description:

- `URL`: Contains the target URL.
- `Cookie`: Contains an authentication cookie for the application.

Model: Given a specific URL of a web application retrieved from the configuration, the attacker model depicted in figure 1 uses the primitive `snd` to send a request to the web server. The server response which returns the content (in this case using the parametrized `rcv` message), is used by the attacker to assign variables used to count the number of forms in the returned page (using the `listForms` and `length` primitives).

In the next step, all fields contained within a form are fetched (using `listFields`). By retrieving the *input* parameter from the instantiation library selected, and embedding that attack string into an input field, an attack request can be generated. Executing such a request returns content from the server. By parsing this content for the corresponding indicator available from the instantiation library (*output*), it is possible to find out whether an attack was seemingly successful.

If this is done in turn for all available forms, fields as well as all possible elements of the instantiation libraries attack string in an iterative manner, an exhaustive list of attack

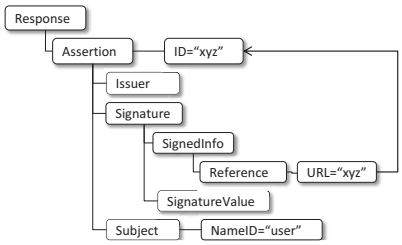

Figure 5. Expected SAML Message Containing Authentication Assertion

requests can be generated. As soon as such a seemingly successful attack has been found, the automata reaches a final state and notifies the security expert. The expert is then able to test this conjecture either by running additional tests against the field, possible by using the same process described with additional patterns from either the same, or a different instantiation library, or by using further manual tests. The process in this case largely depend on the targeted system as well as the information available.

B. XML Signature Wrapping Attack on SAML SSO

XML Signature is the standard protection means for XML encoded messages [7]. The so-called XML Signature Wrapping Attack illustrates that, an adversary may be able to gain unauthorized access to protected resources, by injecting unauthorized data into a signed XML document alongside a possible restructuring in a way that the document's integrity is still verified.

The OASIS *Security Assertion Markup Language 2.0* [8] Web browser Single Sign-On (SAML SSO, for short) is the standard enabling on-line business partners to authenticate their users once within a federated identity environment. Three roles take part in the protocol: a client C, an identity provider IdP and a service provider SP. The objective of C, typically a web browser guided by a user, is to get access to a service or a resource provided by SP. IdP authenticates C and issues corresponding authentication assertions. The SSO protocol ends when SP consumes the assertions generated by IdP to grant or deny C access to the requested resource. The authentication assertion is XML encoded and signed by IdP. Thus, if the signature validation logic of SP is not well implemented, SAML SSO could be vulnerable to XML Signature wrapping attack.

The essential elements in a valid and expected authentication response message containing authentication assertion is depicted in Figure 5. One possible way to perform XML Signature wrapping attack is to put the assertion in the original message inside a “wrapper” node, and create an “attack” assertion whose signature referring to the original assertion. An example of the modified message is depicted in Figure 6. In this case, if the signature validation logic is not well implemented, the signature in the “attack” node will be verified successfully, and “admin” in the “attack” node,

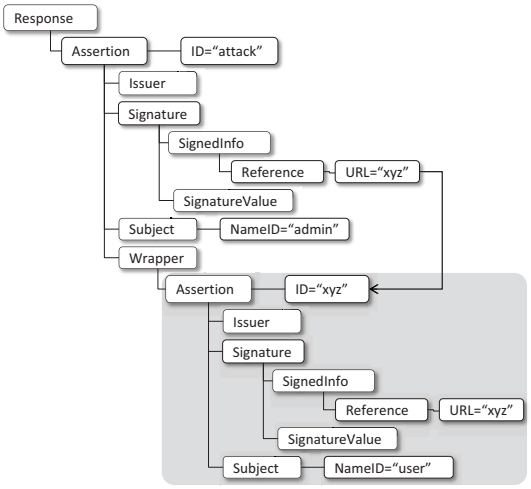

Figure 6. XML Signature Wrapping Attack

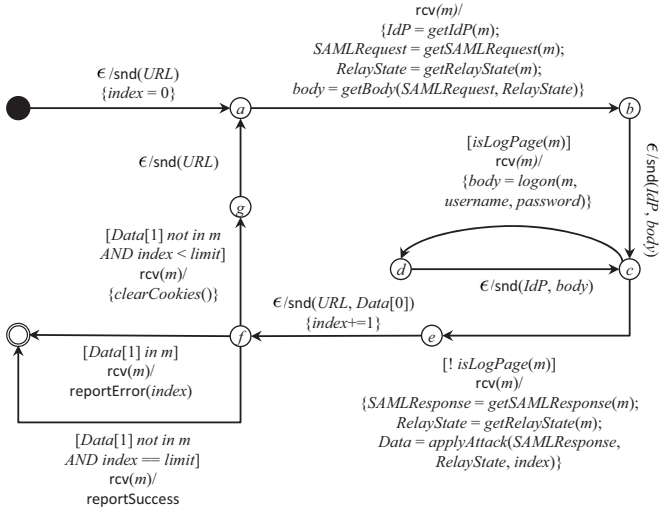

Figure 7. Attacker Model of XML Signature Wrapping Attack

rather than "user" in the original assertion, will be taken as user ID. There are other ways to perform this attack. More detailed discussions could be found in [9].

In order to check whether the signature validation logic is well implemented in SP, we perform testing on SP using attacker model by creating and sending variations of the authentication response message and checking the feedback received from SP. The prerequisite of performing this testing is that we have valid credential in IdP.

The model is depicted in Figure 7. In this model, URL is the address of the SP, and $username$ and $password$ are the credential we have in IdP. With the Vera tool, this attacker model is executed to test several implementations of SAML SSO. The attacker model is executed as follows:

- At first, a normal resource request message is sent to SP (transition from starting state to a).
- From the message received from SP, IdP, SAML Request and Relay State are extracted, and a message to be sent to IdP is constructed (transition $a \rightarrow b$).
- After sending the message to IdP (transition $b \rightarrow c$), if logon page is received from IdP, a logon message is constructed (transition $c \rightarrow d$), and sent to IdP (transition $d \rightarrow c$).
- If the authentication response message is received from IdP, the valid SAML Response and Relay State are extracted, and the "$applyAttack$" function is called to obtain $Data$ (transition $c \rightarrow e$), which is a pair of strings, where $Data[0]$ is the variation of authentication response message, and $Data[1]$ is a "checking string". For example, in the case of wrapping the assertion as depicted in Figure 6, the checking string is the user ID "admin" contained in the attack node.
- Then, $Data[0]$ is sent to SP (transition $e \rightarrow f$).
- If the checking string is found in the feedback received from SP, the testing is stopped and an error is reported. Otherwise, if all of the test cases are executed, the testing is stopped and success is reported (transitions from f to the final state).
- If there are still test cases to be executed, cookies are cleared (transition $f \rightarrow g$), and the initial resource request message is sent to SP to restart the procedure (transition $g \rightarrow a$).

C. Application to a WebGoat lesson

In this section, we will show how to use the VERA tool to perform an SQL-Injection attack based on one of the lessons of the WebGoat[2] application.

After starting a new Vera project, the user is presented with the graphical interface with which he can interact with the tool.

In this case, we want to use a specific attack from the provided library, SQL injection, against our target website. Therefore, the first step is opening the existing file `models/injection.xml` from the available library, resulting in it being loaded into the main window of the graphical interface of the VERA-tool (see figure 9). The window is divided into two areas: i) the editable area where we can modify the models, and ii) the palette that the user can use during the creation of the model.

The user can change the links and nodes' properties through the graphical interface (in figure 8 we show the property tab for a link in the model). Although this will not be necessary in this example, as the provided models and libraries are generic enough to find the instances of SQL injection presented in the WebGoat application.

The configuration file for the WebGoat SQL-injection lesson can also be inserted in the system through a graphical menu. The configuration files are text file containing all

[2]https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

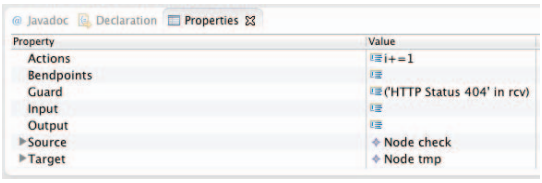

Figure 8. VERA-Tool: link properties

the information needed for the correct execution of the test engine; see e.g. Figure 3 for the configuration file for the WebGoat SQL-injection lesson containing hostname and login information.

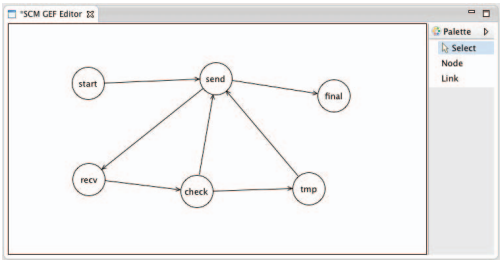

Figure 9. VERA-Tool: State chart diagram graphical editor

Now, it is possible to actually select the instantiation libraries we want. As stated before, VERA can execute multiple Instantiation files, but as we are targeting SQL injections, the generic instantiation file provided suffices (figure 10).

```
IO=[
   ["'","nexpected"],
   ["\'","nexpected"],
   ["1 OR 1=", "nexpected"],
   ["123 OR 1=1--", "Congratulations"],
   ["' OR 1=1--", "Congratulations"]
   ]
```

Figure 10. VERA-Tool: SQL-injection Instantiation File

All that remains to do is execute the tool, resulting in the output shown in figure 11, as well as the WebGoat "lesson complete" page.

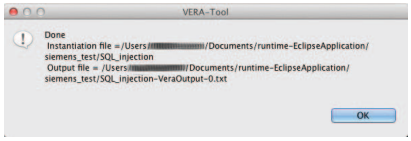

Figure 11. VERA-Tool: plugin output

V. Conclusions

We have presented a tool that allows testers to define and execute models of attackers, aiming at finding software vulnerabilities. Although so far we explored primarily vulnerabilities in web applications, our approach is general enough to be applied in other contexts. There are many directions in which our work could be extended. On the one hand it would be interesting to explore a notion of composition in which multiple attacker models are combined to reach a certain goal. On the other hand, it would be interesting to be able to classify and prioritize tests according to the expected skill level of attackers and the expected impact of vulnerabilities. Currently, preliminary validation steps are being made within Siemens and SAP in order to evaluate the benefits of our approach. Those efforts include using the tool in real-life penetration testing activities. A thorough report on our experiments in this direction will be subject of future work.

Acknowledgments

This work was partially supported by the EU FP7 Project no. 257876, "SPaCIoS: Secure Provision and Consumption in the Internet of Services" (www.spacios.eu) and no. 256980 "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems".

References

[1] Portswigger, "Burp Suite," http://portswigger.net/burp/, 2012.

[2] A. Riancho, "Web Application Attack and Audit Framework," http://w3af.sourceforge.net/, 2012.

[3] SPaCIoS, "Deliverable 2.4.1: Definition of Attacker Behavior Models," 2012.

[4] G. H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[5] J. Tenzer and P. Stevens, "On modelling recursive calls and callbacks with two variants of unified modelling language state diagrams," *Form. Asp. Comput.*, vol. 18, pp. 397–420, November 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1179488.1179491

[6] R. Eshuis, "Reconciling statechart semantics," *Sci. Comput. Program.*, vol. 74, no. 3, pp. 65–99, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2008.09.001

[7] W3C, "XML Signature Syntax and Processing," http://www.w3.org/TR/xmldsig-core/, 2008.

[8] O. Consortium, "Security Assertion Markup Language V2.0 Technical Overview," http://wiki.oasis-open.org/security/Saml2TechOverview, Mar. 2008.

[9] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On breaking saml: be whoever you want to be," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362814