Detecting Attacks against Data in Web Applications

Romaric Ludinard, Éric Totel, Frédéric Tronel

Supélec
Avenue de la Boulaie
35576 Cesson-Sevigné, France
Email: firstname.lastname@supelec.fr

Vincent Nicomette, Mohamed Kaâniche,
Éric Alata, Rim Akrout, Yann Bachy

CNRS, LAAS, 7 Avenue du Colonel Roche, F-31400 Toulouse, France
Université de Toulouse, LAAS, F-31400 Toulouse, France
Email: firstname.lastname@laas.fr

***Abstract*—RRABIDS (Ruby on Rails Anomaly Based Intrusion Detection System) is an application level intrusion detection system for applications implemented with the Ruby on Rails framework. It is aimed at detecting attacks against data in the context of web applications. This anomaly based IDS focuses on the modeling of the application profile in the absence of attacks (called normal profile) using invariants. These invariants are discovered during a learning phase. Then, they are used to instrument the web application at source code level, so that a deviation from the normal profile can be detected at run-time. This paper illustrates on simple examples how the approach detects well known categories of web attacks that involve a state violation of the application, such as SQL injections. Finally, an assessment phase is performed to evaluate the accuracy of the detection provided by the proposed approach.**

I. Introduction

Nowadays, there is an increasing trend toward the development of applications and services allowing end users to remotely interact with their information systems through their Web browser. Such web applications offer all functionalities required by the end-user (e.g., webmail, online calendars, e-banking, ...). On the other hand, such applications also exhibit an increasing number of vulnerabilities that can be remotely exploited by attackers to violate the confidentiality, the integrity or the availability of the services delivered to the end-users.

Several mechanisms have been developed to protect web applications against potential attacks. These mechanisms can be used at the network level (such as applicative firewalls), and at application level (such as input sanitisation techniques). However such prevention mechanisms remain largely insufficient as they do not take into account the semantic of the applications they are supposed to protect, thus providing a rather low protection. Hence, it is important to propose mechanisms that could more efficiently detect attacks affecting the integrity of the application state.

In this paper, we propose an anomaly based intrusion detection approach that focuses on application state violations . This approach is based on the automatic generation of invariants that are discovered during a learning phase and verified during the execution of the application. First, Section II presents previous work about intrusion detection that can be applied in the context of web applications. Section III presents the context and a case study of our work, considering in particular Ruby On Rails applications. Section IV, illustrates on a typical example the ideas we are relying on. Then, we present our approach in a general way (Section V), and show how to apply it to web applications (Section VI). Finally, Section VII shows how the proposed intrusion detection mechanisms have been assessed, in order to demonstrate that they accurately detect the types of attacks we focus on.

II. State of the Art

Most work in the context of web attack detection focuses on abnormal network packets [1] or requests [2] and do not take into account the state of the web application itself.

We believe that application level mechanisms can help improving the detection performance as they are able to take advantage of the internal state of the monitored program. Indeed, they have access to all the internal data structures and algorithms used by the program.

When speaking about intrusion detection at application level, we consider three types of approaches: the first approach focuses on the correctness of the Control Flow Graph of the program such as [3] and [4] and consists in verifying that the actions in the program are executed in a correct order. The second approach focuses on the correctness of the data manipulations during program execution such as in [5] or in [6]. In this case, a data-flow graph is computed prior to the execution that contains, for each data item read by an instruction, the set of instructions that may have written its current value. This data-flow graph is then used at run time to verify the integrity of the process data flow that really occurs. If a vulnerability is exploited in the monitored program that corrupts some data, a deviation from the data-flow graph will be detected the next time this data is read. A third approach, as defined by Sarrouy et al. [7], consists in checking the correctness of the data used by the program during its execution, rather than verifying the consistency of the data flow. This approach permits to detect a different kind of attacks on data compared to the data-flow approach (such as the use of incorrect values in legitimate variables). The application normal profile is modeled as a set of constraints on the data used by the application. This model is built during a two-fold learning phase. First the data items used by the application are logged. Then, a set of invariants is generated by a dedicated tool from the testing field called Daikon [8]. These invariants are then checked at run-time to perform the detection.

All these approaches are relevant in the context of both traditional and web applications. For instance, similar approaches have already been investigated by Swaddler [9] in the context of PHP applications. The authors focused both on the control flow of the web application and on the correctness of the data used by the program. [10] is an extension of this work to JSP. The authors propose to build the invariants characterizing the application normal behavior during a learning phase, using the Daikon tool. Then a model checking approach is used to verify if the invariants can be violated by the application, leading thus to the discovery of vulnerabilities in the application. This interesting approach investigates only invariants on inputs and outputs of the functions executed by a servlet, and does not take into account temporal relationships between the variables used by the program. In our work, we focus on the correctness of data at a finer granularity (i.e., at the instruction execution level) like in [7], and we study the evolution of variables with respect to time so as to discover invariants on these variables. Compared to Swaddler, we additionally introduce the notion of tainted data [11] to reduce the set of variables we are checking. To demonstrate the feasibility of our approach at application level, we consider the example of applications implemented with the Ruby on Rails framework. This framework is receiving increasing interest by the software development community. In the next section, we present the main concepts of Ruby on Rails that are needed to explain our intrusion detection approach, and we present the application example used as a case study in our work.

Compared to previous work, the approach proposed in this paper presents innovative aspects: (1) In all the work dedicated to intrusion detection that uses Daikon, the authors use the available Daikon front-ends or extends them. They thus obtain invariants on inputs or outputs of the methods or functions called. In our approach, we observe the execution of each instruction within the Ruby interpreter and we derive invariants on all the variables processed by the interpreter. Thus, more invariants can be found compared to other approaches such as [9], [11], allowing a better coverage for the detection of attacks. (2) Our approach uses Daikon in a non conventional way, to extract from the execution traces temporal relationships on the program variables, which is a completely new functionality for a work based on Daikon. (3) The implementation we propose does not imply the modification of the interpreter, compared to other approaches, providing thus an elegant implementation of invariant discovery and checking.

III. Context and Case Study

Ruby on Rails[1] (RoR) is a Model-View-Controller framework that eases the way to design web applications. The choice of RoR as a case study was dictated by the fact that it provides introspection mechanisms that make it possible to observe the behavior of the application without modifying the Ruby interpreter. A typical application is composed of several controllers (as shown in Figure 1). A request to an application determines which Rails controller must be used, and which action of this controller must be called. When one of these controllers is called, it interacts with the Model to access the data usually stored in a database, and generates a view that is returned to the user. Since all the requests sent to the application are processed by a controller (more precisely by an action in this controller), we decided to apply our detection model at the controller level.

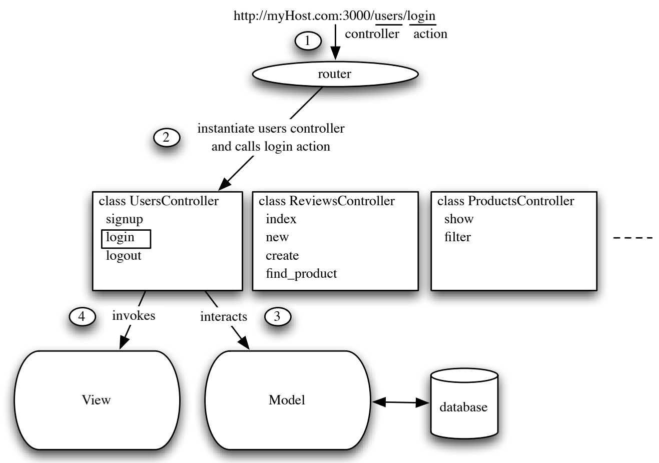

Fig. 1. Rails MVC framework in the context of our application

In this work we applied our approach to an e-commerce application designed by Kereval[2] in the context of the ANR[3] funded project DALI (Design and Assessment of application Level Intrusion detection systems). This application called *Insecure* intentionally exhibits the common vulnerabilities of a web application, including SQL injections, cross-site scripting, request parameter modification and cookie manipulation. It has been designed without any knowledge of the intrusion detection mechanisms that have been subsequently used to protect it. This implies that it was not developed with the objective to emphasize the accuracy of our approach.

IV. Invariant Constraints in an Application

This section shows on a simple example drawn from the *Insecure* application how its normal behaviour can be characterized using invariants on variables, and how an attack can violate at run-time one or more such invariants. We claim that attacks that violate the application state integrity can be detected by checking invariants at run-time.

Figure 2 illustrates the Ruby code used in the application to authenticate a user, located in the action login of the controller Users. Note that the user's login and password are passed as parameters to the action (and stored in the *params* hashtable variable). These parameters are used by a method called *User.authenticate*, that performs a request

[1]http://rubyonrails.org/

[2]www.kereval.com

[3]French National Research Agency

```
class UsersController < ApplicationController
...
  def login
    if request.post?
      # whole user is stored in the session
      if session[:user] = User.authenticate(params[:user][:login],
                                            params[:user][:password])
        flash[:notice]  = 'You␣have␣been␣successfully␣logged␣in.'
        if session[:user].admin
          redirect_to :controller => '/admin/home', :action => 'index'
        else
          redirect_to :controller => '/user/home',  :action => 'index'
        end
      else
        flash.now[:error] = 'Login␣failed'
      end
    end
  end
  ...
end
```

Fig. 2. Login Action in the Users Controller

```
class User < ActiveRecord::Base
 ...
   def self.authenticate(login, pass)
    user=find(:first,:conditions => "login␣=␣'#{login}'␣and␣password␣=␣'#{pass}'")
   end
    ...
end
```

Fig. 3. SQL request in the user Model

to the database to retrieve the actual user corresponding to these parameters (see Figure 3). The normal behaviour of this method consists in returning a user object stored in the session whose attribute login (*session[:user].login*) is equal to the login passed as parameter (*params[:user][:login]*). Thus, the relation *session[:user].login == params[:user][:login]* is an invariant of the application.

If an attacker performs an SQL injection by setting *" 'OR '1'='1"* as user login or password, he will be authenticated as the first user in the database which is in practice the administrator. As a result, the previously mentioned invariant would be violated since *session[:user].login != params[:user][:login].*

This example illustrates a rather simple kind of invariants, that links the output produced by a method call with its inputs. This type of invariants is well known and may even be discovered by static analysis tools [12].

We claim that other types of relationships can be found and should be investigated. In the approaches based on Daikon such as [10], no temporal relationships can be found between successive requests. The authors even claim that such relationships do not exist unless the programmer uses global storage variables such as session variables. In fact, bad programming practice can produce information flows from a request to another, for example by using page parameters. The application *Insecure* exhibits such a practice, in the Order controller. In this controller, the first action to be called is the *new* action that is in charge of presenting a summary of its order to the client. The client must validate all the needed information in the form (a valid postal address, a valid credit card number, etc.). Once the client has validated this form, a second action called *create* is triggered. The amount (price) of this order is transmitted as a parameter of the action. Thus an invariant can be deduced by observing the normal behaviour of the application. Namely, the variable *param[:amount]* in the *create* action must carry the same value as the variable *@amount* got in the *new* action. A possible attack against the application could consist in changing the amount parameter of the second request (using a plugin such as TamperData within the client navigator). However, if the previous invariant had been checked at execution, this attack would have been detected. As far as we know RRABIDS is the first tool to propose this kind of invariants.

V. Invariant based Detection Model

Our work, that builds on the approach by Sarrouy et al. [7] which considered C applications, is based on the dynamic discovery of invariants on traces that are generated during the execution of the program. However, compared to this work, we not only show that the behavior of the program can be modeled as invariants, but we demonstrate that these invariants can be used at execution time to detect attacks. Three types of invariants are considered in this work, to describe: 1) relations between variables at a given execution point; 2) relations between the different values held by a given variable at different points of execution; and 3) relations between different variables at different points of execution.

Clearly, execution points play a central role. We define an execution point p to be a triple (pc, mem, t) that consists of (i) the value of the program counter (pc), (ii) the state of the memory (mem) associated with the program (stack, heap, etc) and (iii) the logical time when this execution point occurs.

It is also necessary to define precisely the notion of a program state. For a given execution and a given execution point, the state of a program consists in the set of all variables together with their associated values that are available at this point (global or local variables). Let σ be an execution of a program. We define $S_{p,\sigma}$ to be the state of this program at execution point p during execution σ. We model $S_{p,\sigma}$ as a partial function from the set of all possible variables to their definition sets. Hence $S_{p,\sigma}(v)$ is the value taken by variable v during execution σ at point p if v is defined or $\perp$ otherwise.

Since we want to discover invariants that are valid for all possible executions, we should not limit ourselves to a single execution (such as σ). Instead we want to consider a set of possible executions that we denote Σ. For two executions σ_1 and σ_2 and two execution points $p_1 \in \sigma_1$ and $p_2 \in \sigma_2$, we say that $p_1 = (pc_1, mem_1, t_1)$ is equivalent to $p_2 = (pc_2, mem_2, t_2)$ denoted $p_1 \equiv p_2$, if and only if their program counters are equal, that is if $pc_1 = pc_2$. Hence, two execution points are equivalent if they correspond to the same position in the program code (same program counter) without taking into account the state of their memories and their occurrence time. We introduce this equivalence relation to be consistent with the method used by Daikon to discover invariants. Indeed, Daikon analyzes traces that contain the values taken by possibly different variables at specific program counters in the program under analysis. Hence, for Daikon two execution points are equivalent as long as they concern the same program counter. More precisely, three kinds of default program counters are defined by Daikon: (1) function entry points, (2) function exit points, and (3) any instruction within the body of a function. Note that in [10], only the first two points are taken into account to compute invariants.

We finally introduce the notion of *unified* state for a given variable v over the set of executions Σ at a given program counter pc to be the function defined by :

$$S_{pc}(v) = \bigcup_{\sigma \in \Sigma} \bigcup_{p \in \sigma | p \equiv (pc, \star, \star)} S_{p,\sigma}(v).$$

The semantic of a *unified* state is as follows: given a program counter pc, a unified state describes the values that can be associated with any accessible variable at this point of the program (that is for all equivalent execution points). From this definition we can deduce the set of all variables V_{pc} at a given execution point $V_{pc} = \{v \mid S_{pc}(v) \neq \perp\}$. This set of variables contains all variables that can be used by Daikon to compute an invariant for a given program counter.

Two points have to be noticed about this model:

- First, it requires to save the values of all the variables each time the program counter evolves. Since this would require a very large amount of memory, we need to reduce the set of variables. This is addressed in Section V-A;
- Second, automatically finding invariants on these variables requires to define what types of invariant constraints are actually needed; this set of constraints is potentially dependent on the model of attacks we intend to detect. This is the purpose of Section V-B.

A. Reducing the Set of Variables

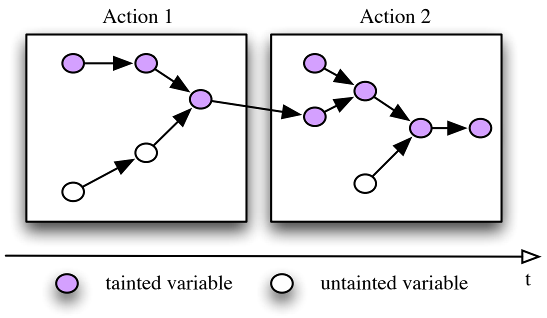

Fig. 4. Variables causally dependent on user inputs

To reduce the set of variables to be logged, we emphasize two considerations:

- First, all variables are not of interest for intrusion detection: an attacker needs to interact with the application, and thus to modify its inputs. The data used by the application are altered by propagation of the modified inputs. As a consequence, the variables that are of interest are those that depend on the external inputs of the application. This set of variables can be defined formally using the notion of causal dependencies.
- Even if we restrict the set of variables to those that depend on external inputs, the application states to be saved are potentially still very large. Consequently, we will only keep in memory all the evolutions of the variables that occur within a given window of time. In this work, we restrict the time window to two consecutive actions performed by a single user identified by its session id (see Figure 4).

Using the notion of causal dependencies as defined in [13], a variable v_1 at time t_1 is causally dependent on a variable v_0 at time t_0 (noted $(v_0,\ t_0) \rightarrow (v_1,\ t_1)$) if there exists an information flow from (v_0, t_0) to (v_1, t_1). The set of variables that influence (v_0, t_0) is called the causality cone and denoted $\mathit{cause}(v_0,\ t_0) = \{(v_i, t_i) \mid (v_i, t_i) \rightarrow^\star (v_0, t_0)\}$ (where $\rightarrow^\star$ is the transitive closure of the relation $\rightarrow$).

This definition can be extended using the execution point previously defined: for a given variable v_0 at point $p_0 = (pc_0, mem_0, t_0)$, with the variable $v_0 \in V_{pc_0}$, $\mathit{cause}(v_0, p_0) = \{(v,p) \mid (v,p) \rightarrow^* (v_0, p_0)\}$, with $(v,p) \rightarrow (v_0, p_0) \Leftrightarrow (v,t) \rightarrow (v_0, t_0)$.

We then define the notion of a *tainted* variable: informally a variable is tainted if it depends on the external inputs of the program (e.g., for a web application, user inputs, data coming from the database or external configuration files). Formally, it means that the causality cone of this variable at a point p of the program contains an input of the program. We can finally define the *tainted* function $T_{p,\sigma}$ of a variable v

$$T_{p,\sigma}(v) = \begin{cases} S_{p,\sigma}(v) \text{ if } v \text{ is tainted} \\ \perp \text{ otherwise} \end{cases}$$

We thus obtain the set of values that can be taken by a tainted variable at a given program counter:

$$T_{pc}(v) = \bigcup_{\sigma \in \Sigma} \bigcup_{p \in \sigma | p \equiv (pc, \star, \star)} T_{p,\sigma}(v).$$

As previously, we can define the set of tainted variables accessible at a given program counter. We denote this set as $TV_{pc} = \{v \mid T_{pc}(v) \neq \perp\}$.

B. Classes of Invariants

Once the set of variable values has been built during an observation phase, we need to automatically generate the set of invariant constraints on this set that characterize the normal behavior of the application. We selected a number of different invariants that could be useful to detect an intrusion:

1) a variable at a specific execution point is always constant; such invariant could be useful to detect the integrity violation of some variables, e.g., some constant values stored in the database to configure the application; this type of invariant consists in finding a relation for a given program counter on all values of v in $T_{pc}(v)$.
2) a variable at a specific execution point holds values that belong to small set of possible values; such invariants could be useful to detect the corruption of some data provided by a user, e.g., a parameter which describes a number of possible options for the execution of a

request; this type of invariant consists in finding a relation for a given program counter on all values of v in $T_{pc}(v)$.

3) a variable at a specific execution point is always equal to another variable at another point; this type of invariant puts in relation some variable values that could be incorrectly modified during an attack, violating the state of the application. For example, the SQL injection described in Section IV violates such an invariant. This type of invariant denotes a link between two sets of variables TV_{pc_1} and TV_{pc_2} where pc_1 and pc_2 are two program counters (pc_1 and pc_2 can be different or equal).
4) an order relationship between two variables at different execution points ($<$ or $>$); like the equality invariant, this relationship describes a normal state of a program that could be altered by an attack that modifies the value of an application variable. This type of invariant denotes a link between two sets of variables TV_{pc_1} and TV_{pc_2} where pc_1 and pc_2 are two program counters (pc_1 and pc_2 can be different or equal).

In practice, complex invariants could be computed on the set of tainted variables, e.g., some linear relations between variables. However, the experience proved us that this restricted class of invariants is sufficient to detect the commonly known web application attacks. The implementation actually shows that these four types of invariants generate a considerable number of actual invariants even on a small application.

The invariants that we discover must not only be true for a given execution σ, but also for all the executions of Σ. In practice, the first executions lead to the discovery of a large set of invariants. Then this set is reduced by invalidating them on further executions of Σ.

VI. Ruby on Rails Implementation

The choice of the Ruby on Rails framework to implement our approach was essentially dictated by the functionalities available in the Ruby language. To implement our model, we need to: (1) intercept the execution of any instruction in the interpreter and (2) access the current execution context (i.e., the variables that are part of this context) at each execution point in order to log the state of the program. The Ruby on Rails framework and Ruby interpreter offer such functionalities without any modification of the Ruby interpreter.

A. Observation phase

Ruby on Rails allows to write plugins for a given application. We wrote a plugin called *trace_vars* to trace and log the state of the application at each executed instruction. This plugin, written in Ruby, is independent of the application source code. It is loaded when the application starts. In an application, all controller classes inherit from an *ApplicationController* class. Thus, the behavior of the whole application can be impacted by modifying the behavior of the *ApplicationController* class. The goal of the *trace_vars* plugin is to interrupt the execution of the interpreter at each instruction executed in a Controller that inherits from the *ApplicationController* class, and to log the application state in a trace. For that purpose, we use the *set_trace_func* function provided by the Ruby interpreter for debugging purposes, to intercept the execution of each instruction in the Ruby interpreter. At each instruction step we log the variables that are in the set of tainted variables previously defined.

1) Observing the program state: In order to obtain the set of variables that are available in the context of the program at a given execution point, we can use the metaprogramming functionalities of Ruby. In particular we can obtain all the instance variables and the local variables of the current executing ruby program by using the instructions $eval(''instance\_variables'', binding)$ and $eval(''local\_variables'', binding)$.

2) Tainted Variables Identification: In order to know if a variable in the state of the application must be logged, we can build its causality cone as defined in Section V-A. This was the approach used in [7]. However, the Ruby interpreter offers an interesting functionality that is available for any object at execution: the notion of *tainted* variables. For each object, a method *tainted?* is available that permits to know if an object is causally dependent on the external inputs of the application. This functionality allows to know if a variable is in the set of tainted variables without having to explicitly compute the dependencies between variables. However, this functionality is only partly implemented in Ruby, as an object can have attributes that are tainted without itself being tainted. As a consequence, when we investigate if a variable is tainted, we have to inspect if any of its attributes are tainted or not. If one of its attributes is tainted, we log this attribute.

3) Naming Variables in the log: To generate invariants on different instances of variables that evolve in time, we must provide to Daikon the values of the variables at different points in time. To offer such functionality, we could introduce additional variables by adding the execution point of the executed instruction to the name of the logged variables. With this approach, we would have the naming rule: *programCounter_VariableName*. However, we must not forget that we will have to instrument the application source code to check for the invariants. As a consequence, we must link the program counter with the line in the source code where the invariant must be checked. To fulfill this constraint, we define the *programCounter* as a couple *(line, numberOfOccurrences)* where *line* is the line executed in the source code, and *numberOfOccurrences* the number of times that this line has been executed. The naming rule of variables becomes: *line_numberOfOccurrences_variableName*. During the observation phase, the variables are stored in a file that is formatted according to the grammar that can be recognized by the Daikon tool. In practice, for each execution of an action, we log the variables in a file. Then, when enough observations have been made, all files are merged into a single Daikon file. This file is in turn used to compute the invariants.

4) Time window handling: In order to be able to find invariants that link variables of two consecutive requests, we need to keep in memory a window of two request executions.

TABLE I
INSTRUMENTATION RESULTS

Subset of *Insecure* files	Initial number of lines	Lines of instrum-ented code	Woven invariants
application_controller.rb	20	20	0
home_controller.rb	11	537	239
products_controller.rb	26	1503	675
reviews_controller.rb	37	1547	691
users_controller.rb	51	1681	771
...	...	...	...

When we log the variables for a given action execution, we actually log the previous execution too. In practice, there can be numerous paths that can lead to a given action execution. In order to be able to discover all relations between a given action and its predecessors, we must execute all possible previous actions. This complicates the learning phase, but it is a testing problem that can be easily solved.

B. Generating Invariants

The invariants are generated with Daikon, a tool commonly used by the software testing community. Daikon is able to generate from a trace file likely invariants on the values held by the variables. The variable values must have been logged in the trace file.

For example, from the file generated for the UsersController and login action, we obtain the invariant shown by Figure 5. This invariant is exactly the one pointed out in Section IV.

```
17_1_@_params["user"]["login"] == 19_2_@_session[:user][:login]
```

Fig. 5. Example of Invariant generated by Daikon

C. Preliminary Results

Even a small application (around 1000 lines of ruby code) and a small class of possible invariant types can lead to a large number of invariants (around 10000). These invariants are woven in the source code (the names of the variables define the point where the invariant must be inserted). As a result, the small source code files containing the controllers of the application can become huge (see Table I for a subset of the application controllers) and the execution time of these controllers increases. We measured execution time for a given set of controller executions. The mean execution time of a controller for non-instrumented source code was about 16.5ms. The mean time for the same executions of the instrumented application reaches 125ms. As a consequence, the instrumented code is about 8 times slower than the non-instrumented one. This is clearly an important difference. However, it must be noted that it remains acceptable for a normal user as the response of the server still seems instantaneous.

VII. INTRUSION DETECTION EFFICIENCY ASSESSMENT

This section presents the experiments carried out to assess the efficiency of the IDS to detect potential attacks, through the assessment of the false negative rate (percentage of attacks seen as valid inputs by the IDS), and the false positive rate (percentage of false alarms).

The evaluation of these measures requires the activation of the application and the IDS under test with a *workload* corresponding to attack-free traffic representative of the various services delivered to the users, combined with a set of malicious traffic (referred to as *attackload*) that should cover various potential attack scenarios.

Sections VII-A and VII-B discuss how the workload and the attackload were generated, respectively. Section VII-C describes the experiments and the results obtained considering the *Insecure* application as a case study. The main conclusions are discussed in Section VII-D.

A. Workload Definition

The workload can be generated with different methods: 1) replay of execution traces collected during "normal" browsing of the web application, 2) use of synthetic traffic generators and 3) real-time browsing of the application. For our experiments, we developed a synthetic traffic generator which offers flexibility to automatically navigate through the different URL links that can be visited from the application under test and to generate various navigation scenarios.

Several web crawlers exist but most of them cannot go through web pages that require prior authentication by providing a valid (login/password) or some specific inputs. Accordingly parts of the application may not be visited. Furthermore, additional functionalities are generally needed to facilitate the generation of attack-free traffic scenarios, such as defining (i) the maximum number of pages to be visited, or (ii) the maximum duration of a user interaction session, (iii) the number of users that can simultaneously interact with the application, or (iv) the time between consecutive user interactions with the application (usually called *think time*). As the crawlers we studied did not provide such a flexibility in their configuration, we developed our own crawler.

B. Attackload Definition

Attackload corresponds to malicious traffic that is aimed at exploiting vulnerabilities of the application. These malicious requests can be generated by hand, or automatically by web *vulnerability scanners* and attack tools.

Many vulnerability scanners have been developed in the last years ([14], [15]). To detect vulnerabilities, they generate specially crafted inputs and analyze the corresponding responses to determine whether the input parameters submitted through the injection point are sanitized or not.

In our experiments, we used `Wasapy`, a vulnerability scanner developed at LAAS, which allows the automated detection and exploitation of different types of web vulnerabilities including SQL injections, OS commanding, File Include, XPath. The algorithm implemented in `Wasapy` is briefly presented in the following considering the example of SQL injections (for more details, see [16]).

The vulnerability detection and exploitation algorithm implemented in `Wasapy` is based on: i) the automatic generation of inputs based on a grammar that is specific to targeted vulnerabilities, and ii) the classification of the corresponding responses using data clustering techniques.

Three sets of requests are submitted at each injection point:

R_{r} correspond to requests with randomly generated data. These are very likely to generate error pages.

R_{ii} correspond to syntactically invalid SQL injections that are designed to lead to unsuccessful executions.

R_{vi} correspond to syntactically valid SQL injections. The main issue is to automatically determine whether they lead to a *successful execution* page or to a failed execution page. To do so, these responses are compared to those associated to R_{r} and R_{ii} using a similarity distance and a data clustering algorithm.

Let us note S_{r}, S_{ii} and S_{vi} the responses associated to R_{r}, R_{ii} and R_{vi} respectively. Then, R_{vi} requests whose responses are not similar to any of the responses from S_{ii} and S_{r} are considered valid SQL injections.

C. Experiments and Results

To evaluate the IDS, we have considered as a case study the *Insecure* application presented in Section III which emulates an e-commerce web site. The users can perform atomic operations such as login to the web site, create an account, browse the list of products, write a review, add a product to a shopping cart, pay, etc. The application contains a few vulnerabilities including SQL injection, XSS, CSRF (*Cross Site Request Forgery*) and FileUpload vulnerabilities.

We have done two sets of experiments. In the first set, only non malicious traffic was submitted to the IDS to assess the rate of false positives under normal use conditions. The workload is generated automatically using the crawler discussed in Section VII-A. In the second set of experiments, we used `Wasapy` to identify the vulnerabilities of the application under test and automatically generate attacks that successfully exploit the identified vulnerabilities. These attacks are complemented by specially crafted attacks, not currently covered by `Wasapy`, that are run manually. During this second set of experiments, both false negatives (lack of detection) and false positives in the presence of attackload only, are identified.

In the following, we provide more details about the two sets of experiments, and the results. The experiments were run on a 3.6Ghz Pentium 4 with 6GB of RAM with Linux 2.6.32, Ruby (version 1.8) Rails (version 2.3.2) and libmsql-ruby 1.8.

1) Experiments with workload only: Our crawler was run during four hours, considering two different classes of users: regular users (80% of the traffic) and admin user (20% of the traffic). Each request is tagged with an identifier to facilitate the processing of the alarms raised by the IDS. Table II summarizes the results of these tests.

20 alarms were raised corresponding to the invalidation of four distinct invariants. They correspond to some normal browsing scenarios that were not activated during the learning phase. For example, one of these invariants asserts that the confirmation password is equal to the password, which in practice is an incorrect invariant. Another one asserts that the credit card of a user who purchases products is always the same, which is also not generally the case in practice.

Number of requests	Number of raised alarms	Number of invalidated invariants
1623	20	4

TABLE II
NUMBER OF INVALIDATED INVARIANTS: WORKLOAD ONLY

Number of crafted requests	Number of successful attacks	Number of raised alarms	false positives	false negatives
13320	11	11	0	0

TABLE III
ATTACKLOAD ONLY

Note that the number of alarms is less relevant than the number of invalidated invariants. Indeed, the number of alarms may be very different from a workload execution to another: if the crawler executes *n* times the scenario in which the password is different from the confirmation password, *n* alarms will be raised that correspond to a single invariant.

2) Experiments with attackload only: For these experiments, attackload was first automatically generated by `Wasapy`. Additionally, some attack scenarios that are not currently supported by `Wasapy` were manually generated.

The clustering algorithm of `Wasapy` was parameterized with 30 requests for each set $[R_{\mathrm{r}}]$, $[R_{\mathrm{ii}}]$ and $[R_{\mathrm{vi}}]$. Thus, 90 requests are sent through each injection point and each attack category. These values are determined empirically, however more requests could also be generated. When the four categories of vulnerabilities implemented in `Wasapy` are investigated (SQL Injection, XPATH injection, File Include and Os Commanding), 90×4, i.e., 360 crafted requests are sent for each injection point. Table III summarizes the results of these tests. `Wasapy` identified *37* injection points and, as a consequence, sent `37`$\times$`360` (i.e., 13320) crafted requests.

The IDS detected all the requests that actually succeeded in exploiting the vulnerabilities. Indeed, among the 90$\times$`37` requests sent by `Wasapy` to detect SQL vulnerabilities, only `11` were successful. The raised alarms exactly correspond to these `11` cases. There are no false positives, neither false negatives. The other requests sent on each injection point, corresponding to XPATH, OS Commanding and File Include attacks, were not successful because such vulnerabilities do not exist in the *Insecure* application. Interestingly, the IDS did not raise any alarm in these cases too.

Three additional attack scenarios were manually executed to exploit some other vulnerabilities: 1) the manual modification of the session cookie (enabling a user to log into the application without supplying a valid login/password, by using the session cookie of an already authenticated user), 2) the manual modification of requests parameters that enables to modify the price of a product during a purchase session and

3) the upload of a Javascript file instead of an image file. The manual modification of the price of a product during a session is correctly detected as it corresponds to the violation of an invariant identified during the learning phase. However, the other attacks were not detected. The upload of a JavaScript File cannot be detected with the proposed IDS because the target of the attack is not the web application itself but the browser (it is a form of XSS attack). As a consequence, it seems difficult to be associated to an invariant. A similar observation can be made concerning the attacks based on the modification of the session cookie. It seems quite difficult to identify one invariant that could be relevant for this kind of attack.

D. Discussion

The experiments described in this section show that the intrusion detection system is particularly efficient to detect successful attacks on the Web application, i.e., crafted requests that actually lead to the integrity violation of the application state. No false positives and no false negatives were generated when the attackload only was sent to the application. Let us emphasize that our vulnerability detection algorithm, sending a large number of crafted requests per injection point, is particularly adapted to evaluate the number of false positives. Regarding the false negatives, some additional experiments still have to be carried out because the *Insecure* application does not include many vulnerabilities. The false negatives must be considered carefully and should be confirmed by testing the IDS on other Ruby applications. Finally, our experiments with attack free workload only illustrate the importance of the learning phase for the intrusion detection system. These experiments raised a few false positives, that can be associated to four invalidated invariants. These incorrect invariants were due to the learning phase, which was incomplete. A more complete learning phase would not have produced these invariants.

VIII. Conclusion

The application level anomaly based intrusion detection mechanisms presented in this paper rely on the modelling of the application normal behaviour using invariants. The violation of an invariant is then considered as an indication of an attack against the application. A first step consists in learning the normal behavior of the application by logging at runtime the value of variables and automatically generating invariants on these data items. The invariants are then woven in the source code to detect a violation of the application normal state during its execution. Although we are looking for a reduced set of invariant classes, the experience proved us that the approach is able to discover a huge number of invariants, even on a small application. Moreover, the assessment phase proved that common attacks, such as SQL injections were successfully and accurately detected by our approach. However some attacks such as Cross Site Scripting or Session Hijacking are not detected by our approach. This is not surprising, as these attacks either impact the client browser, or do not violate the server normal application state. These attacks are thus not of the type of attacks we claim to detect.

References

[1] W. K. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer, "Using generalization and characterization techniques in the anomaly-based detection of web attacks," in *Proc. of the Network and Distributed System Security Symp. (NDSS 2006)*, San Diego, CA, February 2006.

[2] G. Vigna, W. Robertson, V. Kher, and R. A. Kemmerer, "A stateful intrusion detection system for world-wide web servers," in *Proc. of the Annual Computer Security Applications Conference (ACSAC 2003)*, Las Vegas, NV, December 2003, pp. 34–43.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS '05: Proc.of the 12th ACM conference on Computer and communications security*, 2005.

[4] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *Proc. of the Usenix Security Symposium*, 2002.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *2008 IEEE Symp. on Security and Privacy*, 2008.

[6] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *7th USENIX Symp. on Operating Systems Design and Implementation*, 2006.

[7] O. Sarrouy, E. Totel, and B. Jouga, "Application data consistency checking for anomaly based intrusion detection," in *The 11th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, Lyon, November 2009.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, pp. 35–45, 2007.

[9] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications," in *Proc. of the Int. Symp.on Recent Advances in Intrusion Detection (RAID)*, Gold Coast, Australia, September 2007, pp. 63–86.

[10] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *19th USENIX Security Symposium*, Washington, DC, August 2010.

[11] L. Cavallaro and R. Sekar, "Anomalous taint detection," Secure Systems Laboratory, Stony Brook University, Tech. Rep., 2008.

[12] M. Karr, "Affine relationships among variables of a program," in *Acta Informatica*, 1976, pp. 133–151.

[13] B. d'Ausbourg, "Implementing secure dependencies over a network by designing a distributed security subsystem," in *Proc. of the Third European Symp. on Research in Computer Security (ESORICS'94)*, 1994, pp. 247–266.

[14] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *Proc. 2010 IEEE Symp. on Security and Privacy*, Oakland, USA, 2010.

[15] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest : An analysis of black-box web vulnerability scanners," in *Proc. of DIMVA 2010*, 2010.

[16] A. Dessiatnikoff, R. Akrout, E. Alata, M. Kaâniche, and V. Nicomette, "A clustering approach for web vulnerabilities detection," in *Proc. 17th IEEE Pacific Rim Int. Symp. on Dependable Computing (PRDC-2011)*, Pasadena, CA, USA, 2011.