UNIVERSITY OF TWENTE.

EIT ICT Labs Master School
Security And Privacy

Concept-Drift in Web-Based IDS: Evaluating Current Capabilities & Future Challenges

Master Thesis

Nicola Zanetti

Supervisors:
Dr.Sandro Etalle
Dr.Elisa Costante
Dr.Emmanuele Zambon

Eindhoven, August 2015

Abstract

Anomaly based intrusion detection systems (IDS) are typically employed for protecting web applications. Changes in the applications, also known as Web Concept Drifts, negatively impact the accuracy of IDS, increasing the number of false alerts. To adapt the IDS to application changes, the retraining of the model is required. Unfortunately, retraining is a time consuming task that requires a considerable effort from system administrators and security experts. Different methods have been proposed in literature to deal with this issue. One of these, called Response Modeling, exploits the structure of HTTP responses to detect changes and automatically adapt the detection model to application drifts.
In this thesis, we survey existing work that addresses the Concept Drift issue and we test one of them on simulated as well as real scenarios. The results seem to indicate that the existing approach is still not mature enough for consistently reduce the FPR (false positive rate). More precisely, it seems that just a specific type of alerts can be meaningfully reduced while most of the others are not decreased. We propose some requirements and future directions to improve such solutions, aimed at refine the efficacy of this technique.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

Nowadays, web applications are widely used for both individuals and business alike. More and more systems are connected to the world wide web, allowing businesses to reach customers across the globe and providing people with the ability to perform bank transactions, online payments and participate to online social communities. Since online services store sensitive information users wish to protect, there is an increased interest in securing these processes which are constantly subject to cyber attacks. According to Symantec's internet security threat report, there have been more than 6000 total vulnerabilities in 2014, 20% of which were considered critical, meaning they could be exploited to access sensitive data or compromise website content [14]. Table 1.1 shows the top 10 website vulnerabilities exploited in 2014, including SSL/TLS Poodle Vulnerability and Cross-Site Scripting.
Intrusion detection systems, also known as IDS, are used to strengthen the security of web applications. There are two main classes of such tools: signature based and anomaly based. Signature based intrusion detection systems rely on a number of signatures used as patterns to spot existing attacks [26]. Unfortunately, it is challenging to keep signatures up to that with respect to the massive number of continuously discovered attacks. Anomaly based IDS, on the other hand, model the normal behaviour of an application and detect attacks based on the idea that attacks generate anomalies, which are associated with malicious activities. Generally, the behaviour of a web application is defined as a set of characteristics (or *features*) like the character distribution of the parameters processed during normal activities, the name of the parameters and the sequence of resources accessed during a session. Anomaly based systems seem to be unable to adapt to changes, also known as *Concept Drift*, to the monitored applications. Due to a concept drift, the features and functionalities of the application including the structure of the HTTP requests, sessions and responses change due to software updates. When anomaly based detection systems are employed in such circumstances, legitimate interactions between clients and monitored applications might be flagged as anomalous, since the models used for detection do not match their new status. This phenomenon leads to an increased number of false alerts,

Rank	Name
1	SSL-TLS Poodle Vulnerability
2	Cross-Site Scripting
3	SSL v2 support detected
4	SSL Weak Cipher Suites Supported
5	Invalid SSL certificate chain
6	Missing Secure Attribute in an Encrypted Session (SSL) Cookie
7	SSL and TLS protocols renegotiation vulnerability
8	PHP 'strrchr()' Function Information Disclosure vulnerability
9	HTTP TRACE XSS attack
10	OpenSSL 'bnwexpend()' Error Handling Unspecified vulnerability

Table 1.1: Top vulnerabilities on Web servers in 2014 according to Symantec's studies.

jeopardizing the overall detection accuracy. To deal with this problem, the IDS has to undertake a retraining process which is time consuming, and requires a joined effort between developers, system administrators and security experts [31].

1.2 Intrusion Detection Systems

There are currently two main types of intrusion detection systems employed to protect web applications: *signature based* and *anomaly based*. The first class of detection systems is based on a set of signatures of known attacks, which is being updated as new vulnerabilities are discovered. Unfortunately, these systems have to be kept updated frequently which sometimes seems to be difficult considering the amount of new attacks continuously being uncovered and the security expertise these systems require [26]. The second class of intrusion detection systems models the normal behaviour of a web application and detects attacks based on the idea that attacks generate anomalies, which are usually related to malicious activities. Unlike signature based IDS, these tools are capable of self updating and adapting to new attacks, thereby limiting the maintenance and manual updates required [31]. In the next sections, we will provide a more detailed description of both systems and we will give some examples of IDS being used in production environments.

1.2.1 Signature based IDS

Signature based IDS, also known as misuse based IDS, take advantage of existing database of signatures describing different types of attacks. This initial set is specified a priori and updated as new attacks are discovered. An example of such systems is *Snort*, a packet sniffer and logger used as network intrusion detection tool to protect TCP networks. Snort takes advantages of a set of rules, representing signature of known attacks that can be updated using a dedicated programming language. Snort is able to detect and report a

variety of attacks such as buffer overflows, stealth port scans and CGI attacks through a real time alert system [34]. Another commercially available signature based intrusion detection system is *Real Secure* [9], produced by Internet Security Systems. Real Secure is a three layer IDS consisting of a host and network recognition engine and an administrator module. The network and host recognition engines are responsible for detecting attacks that match existing signatures at network and system level respectively. These two engines can also perform intrusion response activities such as terminating a network connection, reconfiguring firewalls, terminating user processes and blocking user accounts. The administrative module handles the data coming from both engines and displays it using a single located administration interface [24, 15, 20]. The system *Shadow* [10], built as a joint project by Naval Surface Weapons Center Dahlgren, Network Flight Recorder, the National Security Agency and the SANS Institute, is a public domain signature based intrusion detection tool which is composed of multiple stations responsible for detection and analysis. The detection (or sensor) stations monitor the network traffic at key locations like outside the firewall, capturing without preprocessing packet headers and logging the traffic information into files that are periodically read by the analysis station. As with Real Secure, the results are displayed in a Web-based interface for further analysis [24, 15, 20].

Signature based IDS systems are effective against specified, well known attacks. They provide lower false positive rates compared to anomaly based ones [33]. However, they are incapable of effectively detecting zero day exploits based on new vulnerabilities, even if they are just minimum variants of existing attacks. Besides, some exploits such as those causing local buffer overflows or those that take advantage of race conditions do not necessarily have a fixed pattern [18]. Moreover, the signature set used by signature based intrusion detection systems contain a considerable amount of entries which might be irrelevant for some organizations. For instance, according to [18], the Snort version 1.8.6 relies on a set of signatures, 516 of which are entries related to web attacks. In case a company used an IIS web server, just a fraction of them would be relevant. The size of such signatures sets might be unreasonably big, thereby undermining the IDS performance.

1.2.2 Anomaly based IDS

Unlike signature based intrusion detection systems, anomaly based IDS build models used to represent the normal behaviour of monitored web applications. Any deviation from these models is interpreted as an anomalous activity and flagged as possible attack. The basic assumption behind this approach is that attacks generate actions that differ from the normal functioning: whenever the difference between modelled behaviour and the observed one exceeds a fixed threshold value, an anomaly alert is raised [26, 31, 33].
Anomaly based intrusion detection systems require a training phase during which the models for designing the normal attitude are built. Different characteristics such as the HTTP traffic being exchanged, the distribution of the parameter characters in a query, the length of the attributes or others based on the sequence of queries are used as training samples [26]. In the work presented in [31], the authors describe an approach for classify-

ing HTTP requests that works by organizing resources representing different modules of a web application. Each of these resources is related to a set of possible queries containing name-value parameters pairs. These parameters are then processed to model the normal behaviour of the application and classify legitimate values. Modeling the length of genuine attribute values, on the other hand, is useful to spot attacks such as buffer overflows and cross site scripting which require an amount of data to be sent significantly higher than the one expected [26]. Besides from HTTP requests, sessions can also be used for modeling the normal behaviour of web applications. Accessing a login page before requesting a private document could be an example of genuine behaviour. If such sequence is modelled as training sample, attacks involving access to private page without authentication could be spotted [26]. An example of session based anomaly IDS is described in [18], where the authors introduce a Session anomaly detection system (SAD) which builds profiles of legitimate page sequences depending on which threshold vales are computed. During detection, any request sequence exceeding these values is flagged as a possible attack. Another example of web anomaly based IDS is described in [21]: the system trains itself using a stream of legitimate SQL queries being exchanged between the web application and the back-end layer and extracts their features, building a representation of their structure called *skeleton query*. During the detection phase, SQL injections are spotted by comparing their query structure with corresponding skeleton queries.

Anomaly based detection systems offer an adaptive solution to web application security. Indeed, they typically do not require maintenance or manual updates to run, unlike signature based IDS where signature databases have to be kept constantly updated by technically skilled personnel. Moreover, unlike signature based IDS, these systems usually support detection of zero day exploits and site specific attacks [26, 31].

1.3 Concept Drift

Today's web applications are dynamic and their content is frequently being updated. Websites are modified frequently, either by GUI changes or by the introduction of new features. These updates modify the structure of the HTTP request served as well as the HTTP responses'. In [31] the authors analysed the frequency of updates of real life websites by monitoring changes in the status of responses and requests. About 40% of the applications considered showed significant updates of page forms during the observation period while about 30% of them changed their input fields.
Additionally, request sessions are also subject to change. The authors in [31], reports drifts in source code repositories of three largest open source applications to verify how frequently code sections responsible for handling HTTP requests sessions are being updated. The research shows that, as with responses and requests, session variations are frequent. Changes in either of these three classes of features lead to a phenomenon known as *Web application Concept Drift* [31]. When anomaly based detection systems are employed to protect web applications frequently being updated, the models used to detect malicious behaviour must be retrained to match the new changes otherwise a considerable amount of false alerts will be triggered. To clarify the reason behind this side effect, let us consider the scenario where HTTP requests are updated in a new version of a web application. There are two possible updates that can affect requests. The fist one, is about modifying the actual value of the parameters, while the second one involves removing or adding a parameter. For instance, a web application has a registration form that handles personal data allowing the user to specify his or her date of birth using a parameter value that supports strings only in a specific format. If a new version of the web application is deployed to support also a different date string format and the anomalous detection system used for protecting the website is not updated accordingly, the system will flag any legitimate request containing these new or modified parameters as malicious. Indeed, the models used for detection rely on some of the methods described in Section 1.2.2, such as strings character distribution or length. If the new deployed application introduces changes in parameters values, these will reflect on the distribution of the characters in the parameters as well as on their length. Therefore, new legitimate requests containing these values will be reported as anomalous [31]. Another scenario that clarifies the importance of models retraining in case of Concept Drift is when the web application being monitored is updated to handle different request sessions. Session drifts take place any time an application is updated to support new sequences of resource path: adding new pages or removing them as well as modifying the behaviour of the system to accept a different order in which the resources are accessed affects the models used to classify legitimate sessions. For instance, an application grants users the ability to read the contents of a forum page at */form-results* only after authenticating into the system at the page */auth*. After some time, the software is updated to allow users to display the content of the forum page even without authenticating. In this new version, legitimate request sessions include */form-results,/auth*. Now, while the probability of this sequence in the previous version of the application was close to zero since users had to authenticate in order to view the forum page, the likelihood of it with the current version of the application

is much higher. This affects the models used for detecting malicious sequences based on the access order of resources called during normal operation [31, 26]. New, legitimate request sessions will be reported as anomalous if the system is not retrained accordingly. Therefore, whenever an application drift arises, it is necessary to retrain the learning models to avoid false alerts caused by misclassification of now legitimate actions. If the models are not retrained to meet the new changes, the system will flag genuine inputs as potentially malicious. Retraining models for anomaly detection is a time consuming process which requires a considerable amount of work by security experts and administrators. That's why having an automated mechanism that detects whether the application has been changed and updates the detection models accordingly is helpful.

1.4 Contribution

In this master thesis, we evaluate one of the techniques that have been proposed to deal with the Concept Drift issue, using both simulated and real scenarios. More specifically, we tested a technique presented in [31] that exploits the content of HTTP Responses to update the learning models and reduce the amount of false positives, also known as "Response Modeling" or simply "Adaptive Learning". We performed different tests using an experimental environment where a monitored web application was being updated by introducing new functionalities. Moreover, we tested the technique with the internal network of a financial institution where the number of false alerts triggered after application drifts is too high for security experts to process.
The results show that, even though the selected technique just partially reduces the overall FPR, is does decrease specific types of false alerts (those related to parameters not in the learning model). As a matter of fact, some of the alerts triggered are actually increased as a side effect of this process. These alerts include those related to parameter values, based on the length and other features of the parameter vales existing in the learning model. In this master thesis, we propose different methods that could improve this approach, by limiting its side effects.

1.5 Thesis Outline

This work is structured as follows. First of all, we will introduce some of the existing techniques that have been proposed to solve the concept drift issue as well as the research question we are addressing. In Chapter 3 we will introduce the experimental environment and describe its architecture. Chapter 4 details the methods used to perform the tests. Chapter 5 describes the testing phase, showing the results achieved with both the simulated and real financial environments, reporting and comparing alert rates obtained with and without the Response Modeling mechanism. Finally, in Chapter 6 we will sum up the work done and discuss new approaches to be considered for future work aimed at improving the solution with respect to security and performance.

Chapter 2

Related Work

Self-adaptive techniques for anomaly based IDS have also been proposed to address Concept Drift in domains not necessarily related to web applications. In [35], the authors propose an innovative machine learning technique for dynamic adaptation of an anomaly based IDS intrusion models based on swarm intelligence: a bio-inspired method which emulates the behaviour of swarms of animals for addressing complex problems. The reason behind this, is that swarms have the considerable ability to adapt to drastic environment changes and keep functioning even if most of its entities no longer exist. Similarly, anomaly based IDS systems like the one proposed in [35], are simple to implement, self adaptable to external changes and robust [25].

Another approach is described in [27], where the authors propose an adaptive anomaly based IDS which takes advantage of Optimized Hoeffding Tree and Adaptive Drift Detection techniques to address concept drift. The Optimized Hoeffding Tree gathers the misclassification rate and false alert rate from the environment and forwards it to the Adaptive Drift Detection module which identifies changes and triggers adaptation. Whenever a drift is detected, the module deletes old observations and retrains itself by running a new classification. The authors also compared the performance of the Adaptive Drift Detection system with other existing detection models including ADWIN Change Detector approach [16] and EWMA Control chart detection method [30], showing that the system proposed is more accurate that the others and has lower false positive rate. The authors in [29] describe how an anomaly based detection system used for detecting changes in code execution flow has been upgraded to support self adaptation to software patches. Whenever a software is updated, the system call behaviour of a program is likely to change from the one monitored during training phase and classified as normal. As usual, the anomaly detection must be retrained to avoid false alerts. The authors in [29] explore a solution that allows the intrusion detection's system call model to detect code drifts based on binary difference analyser. The method exploits an algorithm for converting the execution-graph used by the anomaly detector engine according to the output of the BinHunt binary difference analysis tool [19]. The experiments show that the execution graphs generated with this approach are good approximations of the ones that would have been achieved with normal retraining of the IDS. Table 1 summarizes the methods described above, reporting

Method	Field of Application	Summary
Swarm Intelligence	Network based IDS	Self adaptation is achieved by using an Artificial Fuzzy Ants Clustering (AFAC) mechanism based on swarm intelligence [35].
Optimized Hoeffding Tree	Network based IDS	The technique relies on Adaptive Drift Detection tool which processes misclassification and false alert rate gathered using the Optimized Hoeffding Tree approach. The system identifies concept drifts and reduces false alert rate [27].
Binary Difference Analyzer	System call based IDS	False positives of system call based IDS are reduced by using the output of a binary difference analyzer used to spot differences between software versions and retrain the learning models accordingly [19].

Table 2.1: Techniques used to improve self-adaptive capabilities of different classes of anomaly based IDS.

their respective filed of application.

Concept drift is an issue to solve in order for anomaly based IDS to be effective whenever concept drifts take place. With respect to web applications, some approaches have already been discussed to deal with this problem, including *Response Modeling approach* [31] and *Adaptive Multi-Diagnoser System approach* [22]. The first approach suppresses alerts related to application drifts by exploiting the content of HTTP responses returned by the web application. This is done by parsing the forms and links returned as part of the responses' body and by updating the learning model accordingly. This approach optimizes the time and resources required for retraining the models, which would normally be time consuming.

The second solution proposed relies on an multi-diagnoser framework for building a self adaptive intrusion detection system capable of recognizing when an application has been updated and adaptation is required. This method takes advantage of a multi layer architecture where observations about the system status made by different diagnosers are cross-checked to determine whether the application has drifted or not. In the next sections, we will detail these two approaches and discuss advantages and disadvantages of both.

2.1 Response Modeling Approach

Response Modeling relies on the fact that changes in web applications can be identified by analyzing HTTP responses returned to the clients. HTTP responses can be represented as a series of links Li and forms Fi related to a set of target resources. The anomaly detector system inspects any response returned to the client, extracting and decomposing each

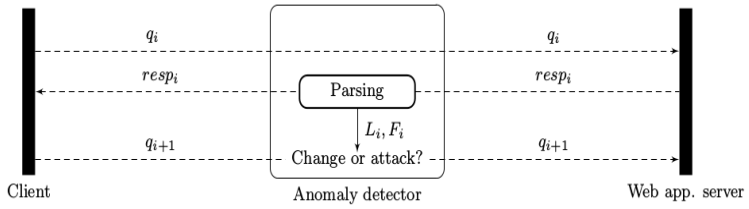

Figure 2.1: Interaction between a client and a web application server, monitored by an anomaly based IDS augmented with Response Modeling. The system detects web drifts by parsing each link Li and form Fi from the HTTP response and by comparing them with the existing reference set. Every time a change is identified, the corresponding models are retrained. Source: [31].

link Li into tokens representing the protocol used, target host, port and path. The same decomposition takes place for forms: for any Fi found, the action attribute is extracted and anlayzed along with any input filed and its parameters. The system processes each response sent by the web application by decomposing it as described above. Each link and form is extracted and its associated values compared with the set built during the training phase. In case a new value or a new parameter has not been previously identified, the learning model is updated accordingly [31]. Figure 2.1 shows the interaction between client and server when Response Modeling is engaged to protect a web application.

Response Modeling can identify when the structure of a web application is updated by introducing new resources, parameters and parameter values. Whenever the application drifts, new models are built to adapt to its new structure. One limitation of this approach is that it relies on the assumption that the web application has not been compromised by an attacker. Since the IDS recognizes changes based on the structure of the documents the application sends to the client, if an attacker had access to the web application being monitored, he could introduce new changes that would be classified as normal updates by the anomaly detection system. The attacker could then exploit the change introduced to vector new threats [31]. The second limitation is that the set of requested pages might not be always available for a given resource. In case the application being monitored supports client side code for creating page content, for example, there is no telling which parameters, forms and links the response will embody since those contents are established dynamically from the client side.

2.2 Adaptive Multi-Diagnoser System approach

The Adaptive Multi Diagnoser System is another approach used to improve the self adaptivity of anomaly based IDS to re-adapt to changes of the application environment. The approach relies on a series of multi diagnosers that constantly report anomalies found by analysing HTTP requests. These observations are processed by a meta-diagnoser that acts as a common agent which verifies whether the observations made by the nodes are contradictory. If this is the case, the meta-diagnoser recognizes that the system has evolved to a new state. Every node of the diagnoser cluster parses the stream of HTTP requests by

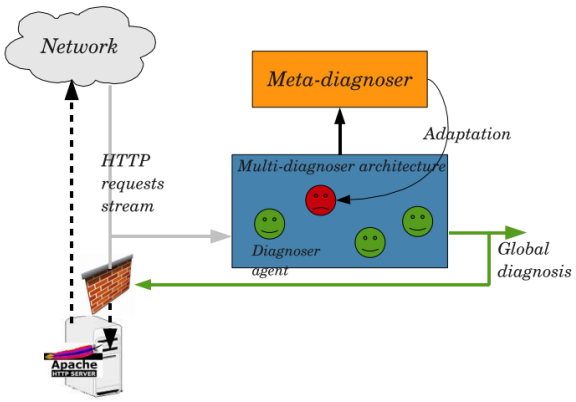

Figure 2.2: Adaptive Multi-Diagnoser System architecture: the multi diagnoser architecture consists of multiple diagnosers periodically reporting observations to the meta-diagnoser. In this example, the observations made by the red agent are inconsistent with the others. As a consequence, the meta-diagnoser might adapt some diagnoser's models. Source: [22]

checking access log files and builds a report of observations taken at different time intervals. The methods for feature extraction are similar to the one discussed in Section 1.2.2, like character distribution and string length. At the same time, the meta-diagnoser compares the reports coming from the cluster agents and builds a global diagnosis, according to which the models used for detection are adapted. Figure 2.2 shows the system architecture. The decision on whether or not updating these models is based on the fact that the reports generated by the cluster agents have to be consistent in time. Indeed, the set of diagnoses should be partially redundant because of the overlapping views of the observes. If this is not the case, then it is likely that some sections of the system environment are drifting and therefore one or more models should be updated to better perform in the future. The same applies in the medical domain, where the status of a patient (like internal temperature or blood pressure) is monitored using different diagnosis tests at the same time. Inconsistency in diagnoses is reason enough to believe that one of the models should be updated [22]. The system, called LogAnalyzer2 [4], has been fully implemented in C++ and tested using a real HTTP server access log of two different research institute collected between 2007 and 2008. In these tests, the authors verified the performance of the system with and without adaptation. The results show that the adaptation substantially reduces the false positives rate. Moreover, the effectiveness of the diagnosers (with respect to sensitivity and precision) seems to be improved when the adaptation is engaged, leading to a better global detection performance.

2.3 Research Questions

The authors in [31] suggest that Response Modeling approach could be useful to reduce the number of alerts generated by anomaly based IDS, when the web application being

monitored drifts. Initial tests have been carried out by the authors, however an in-depth analysis of the solution performance on real scenarios is lacking. Especially, we are interested at evaluating the reduction of false positive in highly dynamic environment which are subject to a considerable number of concept drifts (e.g. twice a week).

In this master thesis we evaluate whether the Response Modeling approach is effective or not at dealing with the Concept Drift issue by performing extensive tests with both simulated and real environments. In particular, we wish to verify its performance on the web application of a financial institution where concept drifts take place frequently, causing an amount of alerts that is impractical to evaluate for security experts. The research question we are addressing in this work is therefore the following:

RQ: *How effective is the Reponse Modeling approach in reducing false positives when deployed in environments where monitored web applications drift frequently and the number of alerts generated is too large for security experts to analyze?*

To answer this question, we will report the results obtained when testing the Response Modeling and discuss whether the alert rate is actually reduced whenever concept drift takes place.

Chapter 3

Solution

In Chapter 2, we discussed some of the solutions that have been proposed in literature to solve the concept drift issue. These solutions seem to be effective (on paper) at reducing the false positives rate generated by anomaly detection engines, whenever web application drifts take place. However, these methods have not been tested deeply on real scenarios, especially those where application drifts take place frequently.

In order to answer the research question we address in this master thesis, we developed the method we believed most promising: the Response Modeling Approach. In this chapter, we will introduce the solution and provide a description of the framework we set up to perform our tests.

3.1 Framework

Our evaluation environment replicates a typical network-based IDS as showed in Figure 3.1. The first step of any IDS is the collection of the network traffic data exchanged between clients and the web application under monitoring. This is done by means of a network analysis tool (*Traffic Sniffer*) that allows the interception (and dump) of network traffic. The network traffic captured by the sniffer is used as main input for our experiments. Note that, as commonly done in the filed of machine learning, a different subset of the network traffic is used for training (*training set*) and testing (*testing set*) activities. Traffic in the training is set parsed and interpreted in order to create a learning model that represents the legitimate behaviour of the web application. The model we use for learning the normal behaviour (*HTTP Learner* in Figure 3.1) is a slight re-adaptation of the approach proposed by Bolzoni et al. in [17]. This model is used by the detection module (*HTTP Detector* in figure) to identify HTTP requests/responses that deviates from normal behaviour, thus can be considered anomalous. In case of anomalous transaction, the detector raises alert that are dispatched to the operator together with information regarding the inputs classified as malicious by the detector.

The aforementioned process is the one typically implemented by any IDS system. In our framework we also account for a module that addresses concept drifts (*Adaptive Learn-*

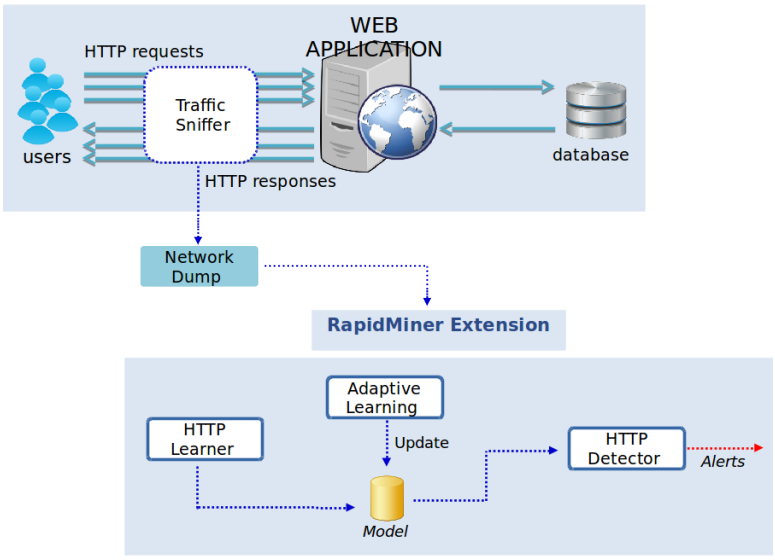

Figure 3.1: Overall system overview.

ing). This module implements the Response Modeling presented in [31] that parses HTTP responses in input and updates the learning model used for detection. Clearly, the detection output (i.e. the alerts) will be influenced by the changes to the underlying learning model. The goal of the module it to influence the alerts in order to decrease false positive. Our goal it so quantify the impact (in terms of reduction of false positive), the Adaptive Learning technique proposed in [31] has on real scenarios.

In the following sections we will introduce the main steps required to set up the environment, including the data collection, learning and detection phases.

3.1.1 Data Collection

The first phase required for the evaluation consists of collecting the network data exchanged between the clients and the web application monitored by the anomaly based IDS. During the learning phase, a network dump file (also called *training set*), is used to create the learning model which will be used to represent the legitimate behaviour of the application. HTTP transactions are parsed from the training set by using a network sniffer which listens to a local interface and collects the traffic exchanged. The same process is used for the detection phase: a network dump is created using a sniffer and the resulting file (*test set*) is parsed and analyzed in the detection phase to spot irregularities. Depending on the detection algorithm being used, transactions obtained from the test file that are not consistent with the learning model are flagged as malicious and trigger an alert.

3.1.2 Learning

As discussed in Chapter 1, anomaly based intrusion detection system take advantage of a learning model created during the training phase that represents the normal behaviour of the web application to detect malicious activity. This model is built by parsing multiple features from the training set, such as the methods, parameters and parameters' values. These are stored in the model and compared against the ones parsed from the the test set. The mismatch between the features in the learning model and those found in the test set is used to generate alerts. The Adaptive Learning mechanism based on Response Modeling also relies on the learning model to reduce the number of false positives when application drifts take place. Indeed, the parameters and parameters' values parsed from the HTTP responses are inserted in the model so that new features implemented no longer raise false alerts.

To build a good approximation of an anomaly based IDS and test the Adaptive Learning efficiency, we had to develop a realistic learning model. Since the authors in [31] do not specify a model structure to be used for the Response Modeling, we built our own based on the work presented in [17]. The model is based on a tree structure – see Figure 3.2– composed of different request's features including *hostname*, *methods*, *parameters* and their values. The hostname is the profile feature that represents the root of the tree model. Different hosts requested would be stored as different trees in the model, each of which would have its own subset of features. The second level of the tree is composed of the HTTP methods being used to request a specific resource, including the POST, GET and HEAD methods. The third level represents all the parameters' name used in the HTTP requests, related to the specific layer-two method they belong to, while the fourth level contains multiple features that are used to represent each parameter and are based on the length of its values. There are multiple types of parameters: Nominal parameters, such as *Username* and *Password*, are used exclusively for string values while Numerical parameters are used to handle integers. There are also Email, Path and URL parameters, used to handle those that have email, path and URL values respectively.
The detection process will use the length of the values (among other criteria) to evaluate the mismatch of a new request (coming from the testing set) with the model. This information is stored in the leaves at the very last level of the tree.

3.1.3 Detection

The detection mechanism takes advantage of the learning model built during the learning phase to identify malicious requests and trigger alerts. The detection uses multiple features, mainly based on the length of the parameter values that have been observed and modelled so far. Alerts can be triggered at different layers of the learning model. With respect to the first layer, any host that was not observed during the training phase and therefore does not exist in the model, triggers an alert. On the other hand, any request method that does match any of the methods stored in the second layer is marked as suspicious. The third layer of the learning model also affects the detection process. Indeed, any

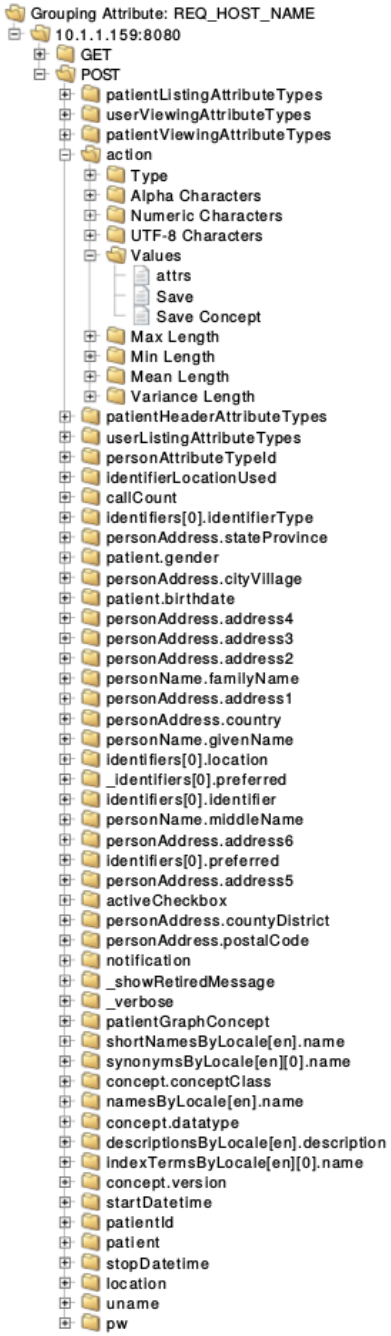

Figure 3.2: Learning model tree built during the learning phase.

request parameter that was not observed in the training phase is also considered malicious: the detection algorithm will flag new parameters as anomalous and will generate an alert. Most of the detection process, however, takes place at the last two layers of the learning

Alert Type	Description
New Method	The request method is missing in the learning model
New Host	The request host is missing in the learning model
New Parameter	The request parameter is missing in the learning model
Suspicious Length	The request parameter value has a suspicious length
Non Alpha Characters	The request parameter value contains non alphanumeric characters
Non UTF8 Encoded	The request parameter value is non UTF8 Encoded

Table 3.1: Alert types.

tree. Here, the length of the parameters' values stored during the training phase is used to identify anomalous activities. More precisely, there are two possible detection mechanisms that can be used in this phase.
The first option is based on the range of values observed so far: the length of the test parameter value is compared against the maximum and minimum length seen so far. Whenever a test value does not fall into this length range, an alert is triggered. For example, a parameter *Username* is profiled with values such as: "Alice", "Bob" and "Jordan". Now, the maximum value length seen so far would be 6, while the minimum would be 3. Any test parameter value that does not fall into this range will raise an alert. Numerical parameters are also profiled the same way: a parameter *Age*, for instance, would be modelled using numerical values representing the age of a user. In this case, the maximum and minimum depend on the actual numerical values rather than their length.
The second detection mechanisms is based on the distribution of the value lengths, and takes advantage of the Chebyshev's inequality [28] to compute the difference between the test length value and the mean of the values observed so far for a specific parameter. This distance is then used to compute a probability which is compared against a threshold value, specified by the user. Probabilities exceeding this threshold are due to values whose length is too far away from the mean, and therefore considered malicious. Clearly, the number of alerts depends directly on the threshold value chosen: the smaller the value, the more alerts are likely to be triggered. For instance, the parameter *Username* described above, would have a mean length value of 4,66. Assuming the threshold value is fixed at 0,01, values too far away from the mean such as "Christopher" or vectors like `<script>alert("XSS")</script>` will raise an alert.
The length of the parameters' values is not the only feature used to trigger alerts. Indeed, the character encoding as well as the presence of only numeric or alphanumeric characters in parameters' values also play a role. With respect to these features, every parameter learned during the training phase is stored in the model with as specific flag used to describe its character encoding as well as whether its values contain alphanumeric or only numeric characters. Any test parameter that has been previously modelled whose values do not match its training reference with respect to these flags, is considered malicious and triggers an alert. Table 3.1 reports the different types of alerts that can be raised during the detection phase.

Chapter 4

Methods

To build our experimental environment we needed to implement the Adaptive Learning techniques suggested in [31] and the normal behaviour method described in [17]. In this Chapter, we will discuss the implementation details of our environment. Especially, we will present the network traffic analysis software used to sniff and inspect the traffic data and some of the modules used to implement the normal behaviour model and the Adaptive Learning.

4.1 Network sniffing

In order to successfully make our tests we had to find a way to parse the network traffic exchanged between web applications and clients. Wireshark [1], is an open source network traffic analyzer typically used to sniff the data exchanged in a computer network. It supports multiple platforms including Linux, OS X and Windows and it is regularly used by professionals such as network engineers, security experts and developers. Unlike *tcpdump* [11], Wireshark has a GUI that allows users to quickly select a network interface to listen to and start monitoring both inbound and outbound traffic. Wireshark allows data capturing live, directly from a live connection and supports multiple network types including Ethernet, Token-Ring, FDDI, serial (PPP and SLIP), 802.11 wireless LAN as well as ATM connections. Wireshark can also capture VoIP calls and play media flows if properly encoded [13]. Wireshark supports multiple data filters and options for dumping the traffic analyzed, which is especially useful for post processing network data: HTTP traffic coming and/or directed to a web server, for instance, can be filtered out and stored in pcap files for further analysis.

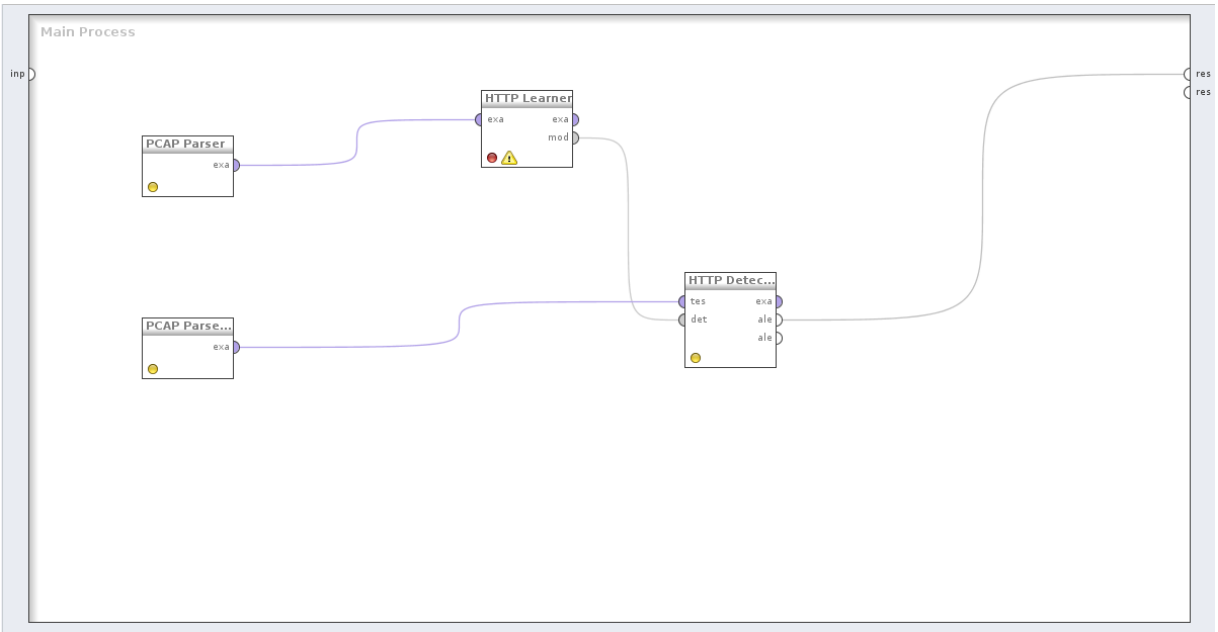

Figure 4.1: RapidMiner process used for learning and detection. As shown in the picture, the HTTP Learner operator creates the learning model out of the training set. The model is then passed to the HTTP Detector operator which compares it with the test set data and generates alerts accordingly. Training and test sets are retrieved from a network dump file previously created with Wireshark, by using the PCAP Parser operator.

4.2 RapidMiner operators

The first step required for evaluating the Adaptive Learning mechanism was to build an anomaly based IDS simulator. For this purpose, we took advantage of RapidMiner [2], a platform with machine learning and data mining support that would allow us to easier the development process. Indeed, rather than implementing a new IDS simulator from scratch, we decided to extend software with existing support for the data processing tasks we required.
RapidMiner is a software developed for machine learning, data and text mining as well as analytics. It has a GUI that allows users to perform data mining operations. It provides tools also called "Operators", that can be used for different purposes, including process control, repository access, modeling as well as data import and export [8]. RapidMiner is fully written in Java [3] and it is also easily extendible: developers can use and modify existing code to customize data processing. RapidMiner is currently used in different fields, from business and education to rapid prototyping [23]. It supports different displaying methods including 3-D graphs, scatter matrices, trees and other charts useful to better visualize processed data.

[1] https://www.wireshark.org/
[2] https://rapidminer.com/
[3] http://www.oracle.com/technetwork/java/index.html

All these features made it particularly suitable for our project, since at least some of the operations required were either already available or extendible from existing sources. Indeed, thanks to RapidMiner's capabilities, we managed to build new software comfortably and integrate it with the existing GUI environment. Nonetheless, we had to implement all the required operators for parsing the network dump pcap files, building the IDS learning and detection process as well as the module required for the Response Modeling. In order to display the data correctly, we also had to develop customized renderers that would allow us to view the learning models as a tree structure. Figure 4.1 shows a RapidMiner process where both the learning and the detection modules are employed. In the next paragraphs we will describe each of the implemented operators.

4.2.1 PCAP Parser operator

This operator is responsible for parsing the network dump files that will be used as training and test set. It takes a file path as input filed, representing the location of the pcap file do be parsed. This operator processes the network data and puts it in an data table to be used for the learning and detection phases.

4.2.2 HTTPLearner Operator

The implementation of the learning operator reproduce the tree structure of the learning model discussed in Section 3.1.2: the root class *HTTPModel* contains the mapping between the *Profile* of the host requested by the client (the root layer of the tree) and the rest of the tree structure, represented by the *ParamModel* class. This mapping is defined as field of the class, of type Java HashMap [2]. Multiple requested hosts would be stored as multiple keys in this map, each of which would have its own *ParamModel* class instance containing the rest of the tree layers. The *Profile* class is used to map Hostnames with the rest of the tree structure. The class ParamModel, on the other hand, maps request methods with the list of their related parameters. Much like the mapping of the HTTPModel class, it is stored as Java HashMap, linking objects of type *Method* (containing the name of the requested method) to the list of *Param* class instances (representing the parameters).

Layer three of the model is implemented by taking advantage of the hierarchy of the Java programming language. Indeed, since the model had to support multiple parameter types including URL, Paths and Emails, we designed a hierarchical structure where the class Param would store basic information about a parameter while its values and corresponding length data would be stored in subclasses. Each parameter is therefore represented as an extension of the Param class and belongs to a specific subclass depending on the nature of the parameter itself. Available subclasses include: *NominalParameter* class, *NumericalParameter* class, *URL parameter* class, *Path parameter* class and *Email Parameter* class. Each of these are composed of multiple fields, used to keep information about values and their respective length. These values are stored in the last layer of the model tree, and are used for the detection process as a template. Some of these features include average, minimum, maximum length of the values seen so far as well as their respective

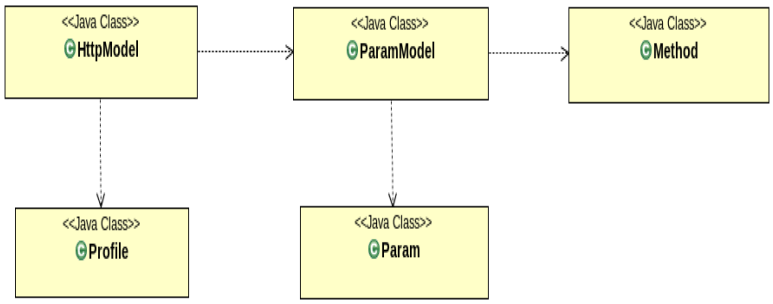

Figure 4.2: Class diagram of the learning process.

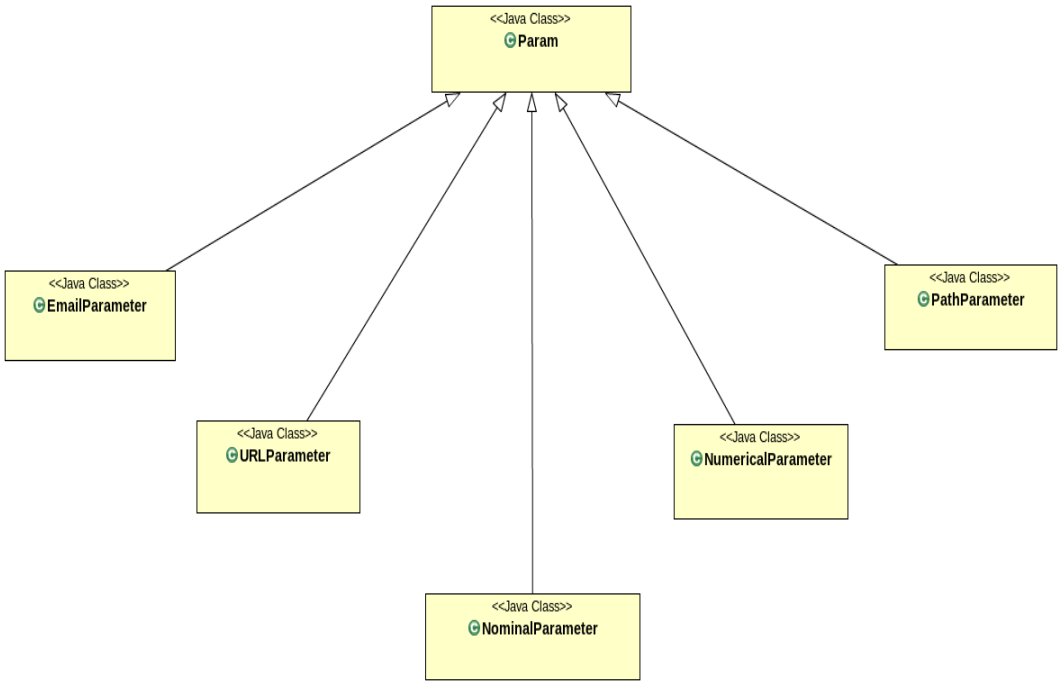

Figure 4.3: Class diagram of the Param class and its subclasses.

variance. There are also some additional fields required for the detection process, including alphanumeric, numeric and UTF-8 flags used to verify whether the parameter handles only values containing alphanumeric or numeric characters or whether their values are UTF-8 encoded. Figure 4.2 illustrates the implementation structure of the learning module, while the hierarchy of the parameter classes is shown in Figure 4.3.

4.2.3 HTTPModelRender Render

The HTTPModelRender is a module used for displaying the HTTPModel created during the learning process. It works by traversing the model tree and displaying all four levels of the model. It visualizes a tree, having the host name as root, method as second layer and

Row No.	Parameter	Anomalous Value	Date	Source IP	User ID	Method	Alarm Description
1	locationWidgetType	tree	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
2	duplicateFormId	1	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	GET	New Parameter
3	name	Peter+Winter	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
4	visitTypesToClose	2	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
5	closeVisitsTaskStarted	on	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
6	enableVisits	on	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
7	_enableVisits	on	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
8	_closeVisitsTaskStarted	on	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
9	_visitTypesToClose	1	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
10	visitEncounterHandler	org.openmrs.api.handler.ExistingVisitAssignmentHandler	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
11	locationId	on	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
12	implementationId	555-555-0199@example.com	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
13	name	Peter+Winter	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
14	passphrase	555-555-0199@example.com	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
15	settings[4].globalProperty.propertyValue	modules	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
16	settings[2].globalProperty.propertyValue	555-555-0199@example.com	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
17	settings[0].globalProperty.propertyValue	en_GB	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
18	settings[3].globalProperty.propertyValue	2	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
19	section	General+Settings	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
20	settings[1].globalProperty.propertyValue	Unknown+Location	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
21	reportType	ATTENDED+CLINIC+THIS+WEEK	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	GET	New Parameter
22	endDate	04%2f01%2f2015	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	GET	New Parameter
23	location	1	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	GET	New Parameter
24	startDate	04%2f01%2f2015	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	GET	New Parameter
25	theme	555-555-0199@example.com	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
26	locale	en_GB	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
27	property	allergy.reaction.ConceptClasses	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter
28	description	A+comma-separated+list+of+the+allowed+concept+classes+for+the+reaction+field+of+the+allergy+	Wed, 03 Jun 2015 13:10:14 GMT	10.1.1.159:8080	NOT AUTHENTICATED	POST	New Parameter

Figure 4.4: Output of the HTTPDetector reporting information about the alerts generated.

their related list of parameters as third. The last layer contains the information about the length of their values, including minimum, maximum, average and variance. An example output of this renderer in visible in Figure 3.2.

4.2.4 HTTPDetector operator

The class that implements the detection process is named HTTPDetector. It supports multiple parameters, raging from the input detection model required for the detection, the test data to be analyzed as well as the threshold value required in case the distribution method is used. The class generates a list of alerts which are stored in a Java ArrayList [3], as the detection process runs. This list is then embedded into a data table by the HTTPDetectionOperator, returned as output. The alerts generated provide information about the source IP, the method, the date, the parameter that caused the alert and its value (if the alert is caused by a parameter), as well as a description of the alert. Figure 4.4 shows an example of alerts triggered by the HTTPDetectionOperator.

4.2.5 ResponseModeling Operator

The Response Modeling operator parses HTTP Responses from a network dump file and updates the learning model passed as input accordingly. The output of this process is a

modified version of the HTTP learning model containing new parameters and/or parameters' values that have been parsed from the HTTP Responses.

As for the implementation, the Response Modeling process is composed of different classes. The *ResponseBodyParser*, *RecorderedHttpResponse* class as well as the *HttpResponseHandler* class, are responsible for handling the network flow and extract the HTTP Responses from it. The class *ModelUpdate*, on the other hand, performs the actual model update by filling it with the parameter-value data that has been extracted from the responses.

Chapter 5

Evaluation

In this Chapter, we will report the results obtained with the Response Modeling mechanism with a simulated dataset as well as with real data. In order to obtain the simulated dataset we set-up an open source web application to which we added new modules to simulate a concept drift. For the real dataset, we obtained real data coming from a financial institution where concept drifts take place frequently (several in a week). The data set for the real case has been obtained by sampling network data several times during more than a month. In the following, we will first discuss the setup, the experiments and the results obtained with respect to the simulation data. Afterwards, we will describe the experimental process executed w.r.t the real scenario at a financial institution. Unfortunately, in this case we will not provide any details regarding the content of the data due to confidentiality reasons.

5.1 Simulation environment

In order to test the Response Modeling approach, we set up a simulation environment composed of OpenMRS [1], an open source web application running on our local environment, an automated web crawler capable of interacting with each form and input field and a network sniffer used to dump and analyze the traffic generated. In the following sections we will describe each of these components.

5.1.1 OpenMRS

OpenMRS is a general purpose open source medical record system. It was developed as software to support healthcare in developing countries. The application supports different systems including Linux, Windows and Mac OS X. OpenMRS is based on a modular structure that allows users to add new features without the need to modify the internal code. The same concept applies for the data used by the system: indeed, "dictionaries" representing data items can be uploaded into the internal MySQL database without manually creating tables or modify the database structure whatsoever [32, 6]. The multi

[1] http://openmrs.org//

modal architecture of OpenMRS makes it particularly suitable for our testing purposes. As a matter of fact, the main requirement for testing the effectiveness of the Response Modeling approach is to verify whether it reduces the false positives rate whenever the application being monitored drifts.
As mentioned earlier, OpenMRS supports multiple modules updates including new forms, input fields and other modifications that actually change the application structure. Therefore, by simply adding new modules to the application, we could simulate application drifts and verify whether the overall alert rate drops after the Response Modeling.

5.1.2 Web Crawler

To generate meaningful network data, we need an automated web crawler tool that would interact with the application by filling forms and other user inputs as well as visiting links. The crawler is used twice during our experiments: (i) to build the training set and (ii) to exercise the web applications after a new module has been installed to simulate the activities generated in case of a concept drift. Wireshark is then used to sniff the traffic on our local network interface and to generate pcap files that will be analyzed using our IDS simulator prototype.
Burp Suite [2] is the crawler we decided to use for our experiments. It is a Java based application that can be used as security test tool to secure or exploit web applications. It is composed of a Proxy server and an Intruder which can be used to automate attacks such as SQL injections, cross-site scripting and parameter manipulation. Burp also provides a web crawler that automatically interacts with the target application by filling forms and following links. The crawler, called Web Spider, can be customized to modify the content to be submitted to forms, and use specific credentials to access restricted pages that require authentication and handle session cookies [1].
The traffic generated by Web Spider can be intercepted by Whireshark server for further analysis, an option that has been particularly useful with respect to our test suite. We also took advantage of the Web Spider's form auto fill capabilities to generate random inputs to submit. This was useful for testing the new functionalities introduced when installing new modules on top of the default OpenMRS application. Indeed, these would likely produce new forms and input fields previously missing. Finally, the data generated is passed to our software prototype for creating the learning model and performing the Response Modeling.

5.2 Simulation results

After downloading the default version of the open source web application OpenMRS, we deployed the WAR file using Tomcat 7 as Servlet Engine and installed some demo data in the MySQL database schema required to run the application [12, 5]. Afterwards, we launched the Web Burp Suite web crawler as well as Wireshark to sniff the traffic on our local interface. The crawler was able to automatically fill out and submit forms with

[2] https://portswigger.net/burp/

random data, besides from visiting links and exploring all the accessible resources.
The network dump obtained was then used in our Rapid Miner extension as a training set to build the detection model. The following step of the test process is to drift the application in order to cause an increased amount of false alerts and compare the alert rate with the one obtained when the Response Modeling is turned on. OpenMRS is built as a modular web application so, in order to generate drifts, we just needed to install new software available on the website rather than modify the sources. These software packages [7] are available online and freely downloadable using an administration panel. In order for this testing to be meaningful, we selected only the modules that are likely to generate changes and raise false alerts, by introducing new forms and forms parameters. These include:

1. **HTML Form Entry Module**: this module allows administrators to create new HTML forms to be filled by users. The forms created are displayed in the "Form entry" section of the patient view.

2. **Jasper Reports Module**: this module generates Jasper Reports. Reports can be customized by specifying name, description, file name of the top level report (if others were previously created). It also allows zip files containing additional reports to be uploaded to the server.

3. **XForms Module**: this module converts normal OpenMRS forms into Microsoft InfoPath forms.

After deploying the modules, we ran the WebCrawler again and started sniffing the traffic as done before for building the training module. We also manually filled out the new forms and visited the links that have been introduced with the last update when the modules were installed, to make sure that a different network traffic would be generated. Once we obtained the training and test set, we executed the detection process to evaluate the false positive rate. Afterwards, we executed the detection process again, but this time using the Response Modeling mechanism to verify whether the number of false positives is actually reduced as expected.

We also noticed that in order to have meaningful results with the Response Modeling mechanism, the test sets used for detection had to satisfy a condition. More precisely, when parsing an HTTP Response, we extract new parameters from forms and links. These parameters will be added to the model of normal behaviour since their presence should not cause a "New Parameter" alert any longer (see Chapter 3). Clearly, to verify if the update to the model impacts the drops of alerts, we need to be sure that HTTP Requests containing these new parameters are contained in the testing set used for the experiment. Therefore, we devised a test to verify the goodness of the testing set. In order to do that, we implemented a class that would count the number n of parameters that are shared between the responses and the requests. If such number is too small, the test set is consider not useful and therefore discarded.

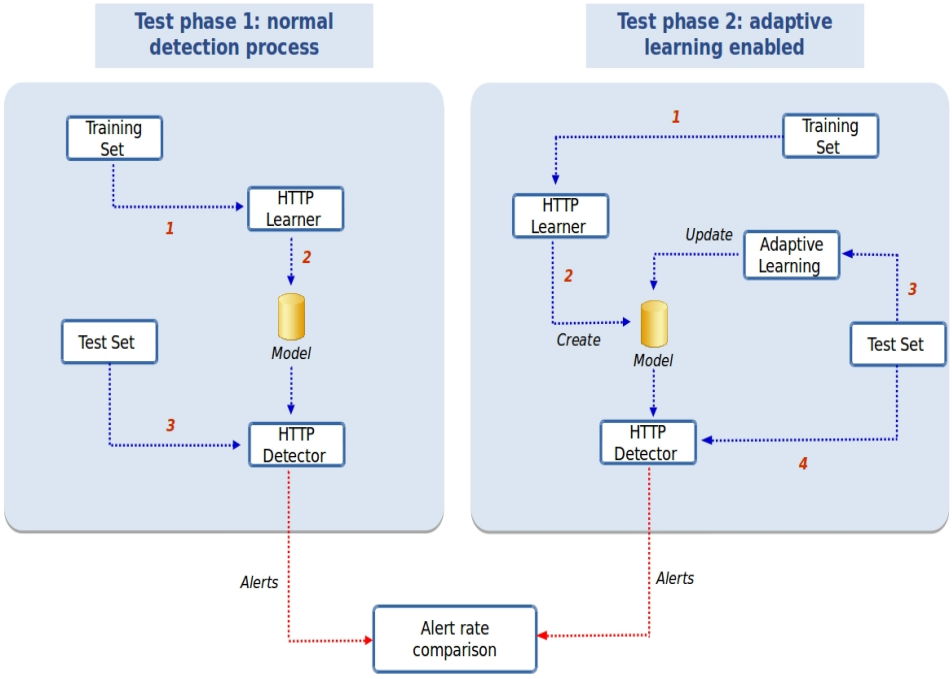

Figure 5.1: Test process workflow.

Formally speaking, given A, the set of new parameters learned from the responses, and β, the set of parameters found in the HTTP requests of the test set, if $A \cap \beta = \emptyset$ then the test set would be discarded, otherwise it would be considered useful.

Figure 5.1 shows the workflow related to the testing phase performed: on the left side, the normal detection process is shown. Step 1 and 2 describe the learning phase were the data from the training test is parsed and used to create the learning model. Step 3 shows the detection phase, where the data obtained from the test set is compared against the learning model to trigger alerts. On the right side of the picture, the Response Modeling process is shown: Step 1 and 2 are exactly the same as with the normal detection process, while Step 3 shows the Response Modeling processing the data from the same test set used for detection and updating the learning model accordingly. Much like with the normal detection process, in Step 4 the detection mechanism compares the test set data with the learning model and triggers alerts accordingly. Finally, the alert rate generated with the normal detection and the one generated with the Response Modeling are compared.

Figure 5.2 and Figure 5.3 illustrate the status of the learning model before and after the Response Modeling mechanism. Listing 5.1 shows a section of an HTTP response that was returned by the application once the new modules have been deployed. Some of the fields of the form contained, including "visitId and "stop Date" were not in the learning model. For this reason, multiple false positive alerts were triggered during the detection phase. The Response Modeling mechanism solved this issue by updating the learning model with these values. Figure 5.3 shows the updated learning model.

```
... <script src="/openmrs-standalone/scripts/timepicker/timepicker.js?v
    =1.10.0" type="text/javascript"></script>
     <div id="patientHeader-endvisit-dialog" title="End Visit">
      <form id="visit" action="/openmrs-standalone/admin/visits/endVisit
         .form" method="post">
       <table cellpadding="3" cellspacing="3" align="center">
        <tbody>
         <tr>
          <td> <input type="hidden" name="visitId" value=""> Enter the
             end date of the visit
    <input type="text" id="enddate_visit" size="20" name="stopDate"
       value="12/11/2014 10:21"
     onclick="showDateTimePicker(this)" readonly><br>   </td>
         </tr>
         <tr height="20"></tr>
         <tr>
          <td colspan="2" style="text-align: center">
    <input type="submit" value="End Visit">  
    <input id="patientHeader-close-endvisit-dialog" type="button" value=
       "Cancel"> </td>
         </tr>
        </tbody>
       </table>
      </form>
     </div>
     <script> ...
```

Listing 5.1: Section of HTTP response body.

In our experiments, we ran three different tests: in the first one we enabled only one of the OpenMRS modules described above and interacted with the application only manually. The alert rate obtained with the normal detection process is of 59% out of 137 HTTP transactions processed from the test file. With the Response Modeling mechanism enabled, this rate drops to 29%. In other words, the total number of alerts to be reviewed is reduced from 81 to 40. The second test performed, is also successful. In this case, we installed two modules of OpenMRS previously described and we ran the web crawler to generate more traffic and fully interact with the new version of the web application. In this case, the total number of HTTP transactions is 401. The detection rate with this test set is reduced from 16% to 7,4% thanks to the Response Modeling mechanism, meaning that the total number of alerts drops from 67 to 30. The last test was performed with all three modules enabled and using the web crawler to automatically generate traffic. This was done to simulate a radical change of the web application, a situation that would take place in case of a full update. In this case, the alert rate is decreased from 44% to 19% while the number of alerts triggered is reduced from 310 to 132.

Test case	Transactions	Rate before	Alerts before	Rate after	Alerts after
1	137	59%	81	29%	40
2	401	16%	67	7,4%	30
3	635	44%	310	19%	132

Table 5.1: Alert detection rates with and without Response Modeling, obtained with the simulated dataset.

These results show that, with respect to the simulation environment we used for our tests, the Response Modeling mechanism is able to successfully reduce the number of false positives in case the application being monitored is updated. The results of the experiments are summarized in Table 5.1.

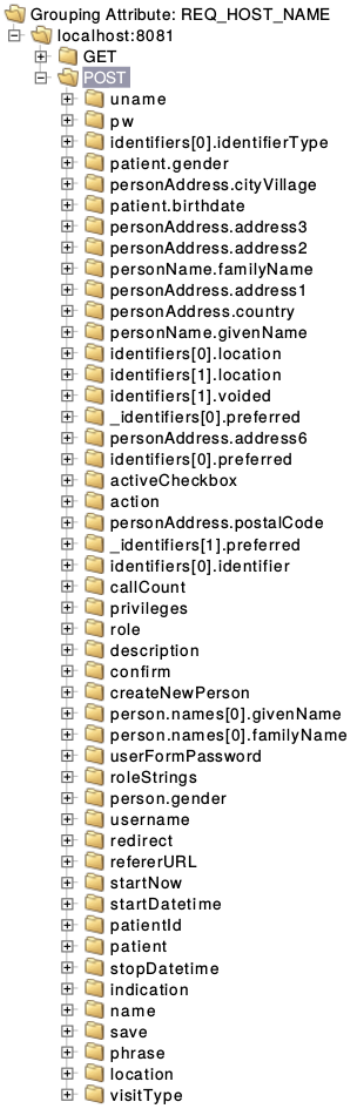

Figure 5.2: Learning model obtained without the Response Modeling.

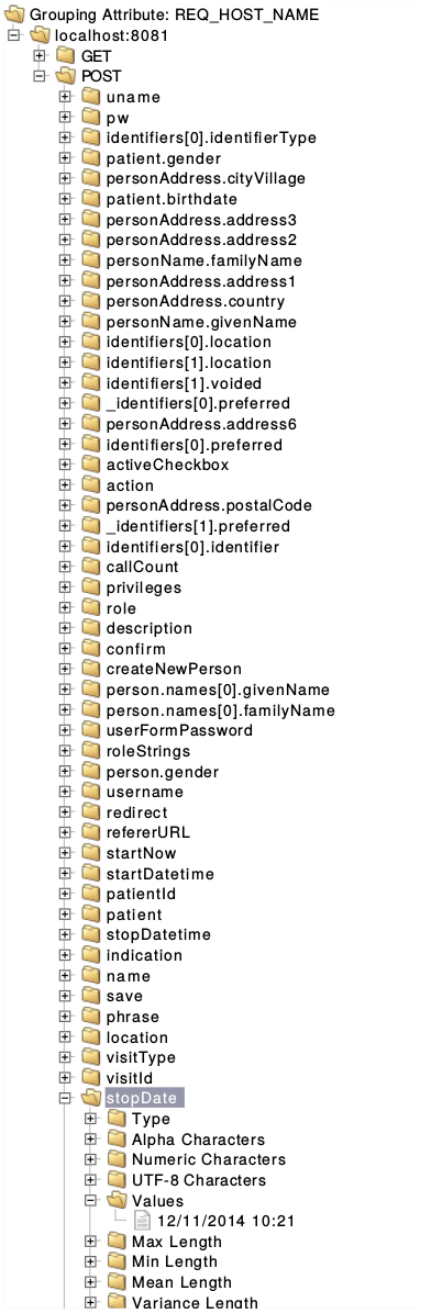

Figure 5.3: Learning model obtained with the Response Modeling.

5.3 Financial case results

In order to better verify the effectiveness of the Response Modeling mechanism, we also ran tests at a financial institution were an anomaly based IDS was deployed. These tests were performed under the supervision of a security expert who helped us gathering the network data and performing the learning and detection processes. We were also being told that the application monitored was updated frequently during the analysis, which was required in order to have a meaningful test set. Due to the confidentiality of the information contained in the training and testing data, we will not discuss neither the details related to the content of the detection models created during the learning and Response Modeling processes, nor the name of the financial institution that helped us with these tests. However, we were given the opportunity to disclose the information useful to verify the effectiveness of the Response Modeling, such as the detection rates, and the number and type of alerts triggered with and without it.

As with the simulation environment, the first step of our tests is to extract the network data needed to build the learning model. Once the learning model is done, we can move ahead with the detection process. Once obtained the detection rates, we turned on the Response Modeling and used the same test set used previously for detection to see whether filling the learning model with responses' input fields could help reduce the false positives rate. In order to have a meaningful alert drop, we tested the test files as we did for the simulation environment to make sure that the parameters learned from the responses would also be found in the requests: network dumps that did not satisfy this condition would not be considered suitable for the evaluation and therefore discarded.

By looking at the results obtained with the process, we noticed that the total amount of alerts dropped just by 8%, after the Response Modeling. This first surprised us, then it led us to believe that there must be other reasons behind it. Indeed, if we look at the drop of alerts only w.r.t. "New Parameter", we see that the total drop is around 20% or more. We believe this reduction in "New Parameter" alerts actually corresponds to an increase of other types of alerts as showed in Figure 5.4. Here we can see that e.g. the number of value based alerts such as "Suspicious Length", "Non Alpha Characters" and "UTF-8 Encoded" are increasing due to the fact that the new parameters recently introduced in the learning model do not have enough reference values to be used during the detection. Indeed, most of the values related to new parameters learned from forms and links cannot be found in responses' body (such as the ones defined as input fields), which leads to an increased amount of value based alerts.
The key observation here is that, the Response Modeling mechanism is effective at reducing specific types of alerts, rather then the overall amount. To cope with this issue, an additional learning phase could be performed to train the new parameters for a period of time. This would fill the learning model with enough values to be used as reference during the detection phase. We will introduce this approach in the next Chapter as part of the Future work.

Test	IS	Transactions	Rate before	Alerts before	Rate after	Alerts after	Alert drop
1	71	20304	9,64%	1958	8,44%	1714	12,46%
2	46	30495	10,02%	3056	9,46%	2886	5,56%
3	91	45147	8,81%	3978	7,55%	3410	14,27%
4	95	45112	9,65%	4352	8,59%	3875	10,96%
5	43	29148	9.60%	2797	8.96%	2613	6,57%
6	91	50280	15.30%	7691	14.89%	7486	2,66%
7	91	58791	14.57%	8566	14.06%	8265	3,51%

Table 5.2: Alert detection rates with and without Response Modeling enabled, obtained with the internal network of a financial institution.

Test	Transactions	NP Alerts before	NP Alerts after	NP Alert drop
1	20304	1233	896	27,33%
2	30495	1328	1167	12,12%
3	45147	2192	1510	31,11%
4	45112	2252	1647	26,87%
5	29148	942	702	25,48%
6	50280	2394	2006	16,21%
7	58791	4124	3656	11,35%

Table 5.3: "New Parameter" (NP) alert rates, obtained with and without Response Modeling enabled on the internal network of a financial institution.

Table 5.2, shows the results obtained with respect to the total number of alerts along with information regarding the total number of HTTP transactions and the size of the intersection set (IS) based on the parameters found in both HTTP responses and requests, used to verify whether the test case could be considered meaningful or not. Table 5.3, on the other hand, shows the performance of the Response Modeling technique with respect to the number of "New Parameter" alerts.

The results indicate that the Response Modeling does reduce the total number of alerts, however it is most effective at limiting those related to "New Parameters". As discussed earlier, this seems to be related to the fact that the new parameters inserted in the learning model now trigger value-based alerts such as the ones discussed in Chapter 3, including "Suspicious Length", "Non UTF-8 Encoded" and "Non Alpha Characters". With respect to "New Parameter" alerts though, the Response Modeling substantially decreases the number of false positives. As shown in Table 5.3, the average drop of these alerts is around 20%.

Figure 5.4 shows the distribution of the alerts by type, obtained with and without the Response Modeling. As shown, the number of "New Parameter" alerts is decreased thanks to the auto tuning. However, alerts based on parameters' values such as "Non Alpha Characters", "Non Numeric Characters" and "Suspicious Length" grow due to the side effect of the Response Modeling mechanism.

Alert Type	Before	After
New Parameter	2192	1510
Non Alpha Characters	365	375
Non Numeric Characters	365	375
Non UTF8 Encoded	80	88
Suspicious Length	976	1062

Table 5.4: Alert drop by type obtained before and after Response Modeling.

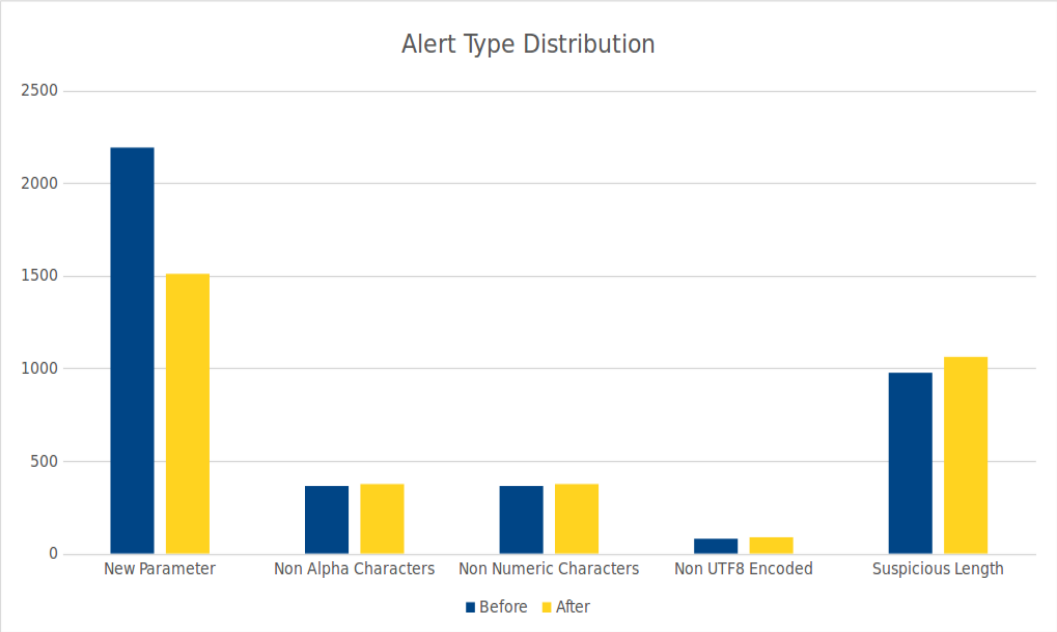

Figure 5.4: Alert Type distribution before and after Modeling: the distribution is based on the data reported in Table 5.4. As shown, even though the number of "New Parameter" alerts is decreased thanks to the Response Modeling, the number of value based alerts grows as a result of its the side effect.

Chapter 6

Conclusion and Future work

Today's web applications process personal data and other sensible information that needs to be secured against web attacks to avoid any leakage or malformation. Anomaly based intrusion detection systems (formally IDS), provide an effective protection against these threats, by modeling the normal behaviour of web applications based on which malicious activities are reported as alerts. However, these IDS are by default unable to deal with changes in the system environment being protected. This issue, also known as Concept Drift, leads to increased false alert rates and worsens the detection accuracy. Manual retraining of the IDS models requires substantial effort and time, therefore an intrusion detection system capable of self adapting to environmental changes is highly desirable.
In this thesis, we have implemented and tested one of the methods proposed to address Concept Drift, using both simulated data and real data obtained from a financial institutions internal network. The method, called *Response Modeling* [31], relies on the structure of HTTP responses to detect changes. Each response issued by the application is parsed to spot previously unclassified parameters which are used to retrain the detection models. According to the results obtained, this approach (also known as Adaptive Learning) does reduce the detection rate whenever a web application is updated. However, while the improvement over the overall alert rate is somehow limited, it seems that the approach effectively reduces the number of false positives with respect to a specific type of alert. This is due to the fact that filling the learning model with new parameters parsed from HTTP responses does not necessarily reduce the whole alert rate. Indeed, most of these parameters cannot be associated with a specific value: for instance, parameters used in some forms would have user-defined data for their input fields. These parameters will therefore be learned without an exhaustive set of values to be used as model for the detection process, which would then flag most of their values as malicious, when analysing the requests. This leads to multiple parameter-values alerts, thereby increasing the overall alert rate. However, the Response Modeling seems to be effective at reducing a specific type of false positives: those related to new parameters. Indeed, the results obtained with the internal financial network show that the average alert drop with respect to these type of alerts has been decreased by c.a. 20%. This proves that, even though the mechanism just partially reduces the overall amount of false positives, it does decrease those related

to new parameters.

The Response Modeling approach seems effective at dealing with concept drifts in web applications. However, it does have some weak points to improve both performance and security wise. With respect to efficiency, parsing HTTP responses returned by the web server might be costly considering the number of requests made by clients to large scale servers. Besides, keep filling the learning model with new parameters and/or parameter values might also jeopardize the overall efficiency of the detection process.
Security is also another concern when it comes to parsing HTTP responses. As a matter of fact, the values parsed from the responses found in forms and/or links, might not necessarily be trustworthy. Indeed, if an exploit is introduced by an attacker into the web application some of the returned HTTP responses might embody threats. For instance, a link containing a SQL injection or a XSS could be returned to the client as part of the response content. The Response Modeling process as it is, would parse such malicious value from the HTTP response and accordingly update the learning model which would label as normal undesired values.

Moreover, as discussed earlier, the Response Modeling mechanism seems to have a side effect on the number of alerts being triggered. More precisely, parameter values based alerts might be increased as a result of the auto tuning process. A possible way to avoid such scenario would be to trigger an additional learning phase, focused specifically on those parameters parsed from responses whose values are not known yet. In the next sections we will introduce some approaches that might be helpful for solving the performance and security issues we discussed, to be considered as future work.

6.1 Future Challenges

6.1.1 Alert Rate Based Optimization

Even though the Response Modeling approach seems effective at reducing the number of false alerts during application changes, it might decrease the overall performance of the detection process. Indeed, parsing each HTTP response returned by the server and updating the learning model seems quite expensive in terms of resources and processing power required. Since the Response Modeling process and the detection would have to take place simultaneously to provide a continuous learning, the overall system performance might be jeopardized. In order to reduce the performance impact of the Response Modeling approach, we propose an optimization that takes advantage of the number of alerts generated during a specific time window. The main observation behind this optimization is that whenever an application drift takes place, the total number of alerts generated is likely to increase. As we discussed in Chapter 1, web drifts cause a large amount of false positives due to the mismatch between the learning model and the current status of the application. An increased amount of false positives also increases the overall number of alerts generated. Therefore, an higher number of alerts in a limited time interval could likely be caused by

an application drift. Since the Response Modeling should be run mainly after application changes take place, there is no need to continuously parse HTTP responses. By evaluating the current alert rate and comparing it with a fixed threshold value (to be estimated according to the average rate), we could tell whether an application has likely been updated and therefore whether a retraining is necessary. This approach would limit the amount of parsing required, since the Response Modeling would only run at specific time periods. Clearly, an increased number of alerts might not necessarily be related to a drift. Indeed, the application might actually have been heavily attacked in a short period of time. In such a scenario, the Response Modeling would be called anyway and the learning model retrained even though there is no actual update. This however, would impact the overall performance only moderately compared to the normal Response Modeling process where each response is parsed no matter the status of the application being monitored.

6.1.2 Signature based optimization

Response Modeling can also be improved with respect to security. Indeed, as we described in Chapter 2, this approach is based on the fact that learning parameter and parameter values found in HTTP responses can improve the detection model reducing the overall amount of false positives when application drifts take place. However, it assumes that each parameter and parameter value parsed from the responses is legit. This might not always be the case. As a matter of fact, the web application being monitored might be compromised by an attacker who could vector an exploit such as XSS, or SQL injection by means of a vulnerable parameter. This would then be reflected in the HTTP response returned by the web server whenever the resource containing the exploit link or script is requested. In such a scenario, the Response Modeling approach will parse such responses and update the learning model with the contained malicious value, thereby compromising the detection process. In order to avoid such a thing, the approach can be augmented with an additional security module responsible for verifying the data parsed from the responses before updating the learning model. This can be done by using a database of attack signatures as support.
By applying this mechanism, corrupted HTTP responses can be detected before updating the learning with malicious values that can compromise the detection accuracy.

6.1.3 Incremental learning

In order to cope with the increased amount of parameter values based alerts caused by the Response Modeling process, we propose an incremental learning approach were an additional learning phase is performed over the parameters that have been introduced during the auto tuning phase. The reason why value based alerts are increased after the auto tuning phase is that, the new parameters parsed from the HTTP responses and included in the learning model do not have an exhaustive set of values to be used as reference for the detection phase. Therefore, a considerable amount of false positives is generated when comparing the test data against the learning model, since most of the

values analysed for those specific parameters are flagged as malicious.
In order to deal with this issue, an additional learning phase could be used to update the learning model with legitimate values. Recently introduced parameters can be held in a sort of *limbo* status for a certain period: during this period when they are intercepted they are used to do learning rather than detection. As a consequence, the detector will not trigger as many alerts related to suspicious values as it would without the additional training.

Even though this approach might help cope with the negative side effects of the Response Modeling approach, it has some drawbacks. Indeed, if an attack is performed during the *limbo* phase on any of the new parameters, malicious values can be interpreted as legitimate causing a deterioration of IDS effectiveness. For this reason, we believe that such values would have to be checked against a database of attack signatures, before being inserted into the learning model: only those values that do not match any of the existing attacks would be considered trustworthy and used for the incremental learning.

Bibliography

[1] BurpSuite. http://portswigger.net/burp/. Accessed May-2015. 26

[2] HashMap. http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html. Accessed May-2015. 21

[3] Java Arraylist. https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html. Accessed May-2015. 23

[4] LogAnalyzer. http://www.irisa.fr/dream/LogAnalyzer/. Accessed May-2015. 10

[5] Mysql. https://www.mysql.com/why-mysql/. Accessed May-2015. 26

[6] OpenMRS. http://openmrs.org/about/faq/. Accessed May-2015. 25

[7] OpenMRS Modules. https://modules.openmrs.org/#/search. Accessed May-2015. 27

[8] rapidmineroperators. http://docs.rapidminer.com/studio/operators/rapidminer-studio-operator-reference.pdf. Accessed May-2015. 20

[9] RealSecure. https://www.enisa.europa.eu/activities/cert/support/chiht/tools/iss-realsecure. Accessed May-2015. 3

[10] Shadow. http://www.securityfocus.com/tools/69. Accessed May-2015. 3

[11] Tcpdump. http://www.tcpdump.org/manpages/tcpdump.1.html. Accessed May-2015. 19

[12] Tomcat7. https://tomcat.apache.org/index.html. Accessed May-2015. 26

[13] Wireshark. https://www.wireshark.org/faq.html. Accessed May-2015. 19

[14] Symantec Internet Security Threat Report. https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf, 2015. Accessed May-2015. 1

[15] L.T. Heberlein B. Mukherjee and K.N. Levitt. Network Intrusion Detection. 1994. 3

[16] Albert Bifet and Ricard Gavalda. Learning from Time-Changing Data with Adaptive Windowing. 2007. 7

[17] Damiano Bolzoni and Sandro Etalle. Boosting web intrusion detection systems by inferring positive signatures. In *On the Move to Meaningful Internet Systems: OTM 2008*, pages 938–955. Springer, 2008. 13, 15, 19

[18] Sanghyun Cho and Sungdeok Cha. SAD: web session anomaly detection based on parameter estimation. 2004. 3, 4

[19] Michael K. Reiter Debin Gao and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. 2008. 7, 8

[20] D. Anderson et al. Detecting Unusual Program Behaviour Using the Statistical Component of the Next-Generation Intrusion Detection Expert System. 1995. 3

[21] Darren Mutz Fredrik Valeur and Giovanni Vigna. A Learning-Based Approach to the Detection of SQL Attacks. 2005. 4

[22] Thomas Guyet, Ren Quiniou, Wei Wang, and Marie-Odile Cordier. Self-adaptive web intrusion detection system. 2009. 8, 10

[23] Markus Hofmann and Ralf Klinkenberg. RapidMiner: Data mining use cases and business analytics applications. 2013. 20

[24] Alan Christie John McHugh and Julia Allen. Defending Yourself: The Role of Intrusion Detection Systems. 2000. 3

[25] Georgios Kambourakis Kolias, Constantinos and M. Maragoudakis. Swarm intelligence in intrusion detection: A survey. 2011. 7

[26] Christopher Kruegel, Giovanni Vigna, and William Robertson. A multi-model approach to the detection of web-based attacks. 2005. 1, 2, 3, 4, 6

[27] S. Ranjitha Kumari and P. KrishnaKumari. Adaptive Anomaly Intrusion Detection System Using Optimized Hoeffding Tree and Adaptive Drift Detection Method. 2014. 7, 8

[28] Kvanli, Alan, Robert Pavur, and Kellie Keeling. *Concise managerial statistics*. Cengage Learning, 2005. 17

[29] Debin Gao Li, Peng and Michael K. Reiter. Automatically Adapting a Trained Anomaly Detector to Software Patches. 2009. 7

[30] et al. Lowry, Cynthia A. A Multivariate Exponentially Weighted Moving Average Control Chart. 1992. 7

[31] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a Moving Target: Addressing Web Application Concept Drift. 2009. 2, 3, 4, 5, 6, 8, 9, 10, 14, 15, 19, 35

[32] et al. Mamlin, Burke W. Cooking up an open source EMR for developing countries: OpenMRSa recipe for successful collaboration. 2006. 25

[33] G. Macia-Fernandez P. Garca-Teodoro, J. Daz-Verdejo and E. Vazquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. 2008. 3

[34] M. Roesch. Snort, lightweight intrusion detection for networks. 1999. 3

[35] Surat Srinoy. An Adaptive IDS Model Based on Swarm Intelligence and Support Vector Machine. 2006. 7, 8

[36] Turner, D., Fossi, M., Johnson, E., and et al. Symantec Global Internet Security Threat ReportTrends for July-December 2007. 2008.