POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Detecting Anomalies In Enterprise Network Events

Relatore
prof. Antonio Lioy

Angelo Mirabella

Supervisore aziendale
Lastline, Inc
dott. ing. Corrado Leita

Anno accademico 2019-2020

Summary

Web based vulnerabilities have been of great interest because of the huge quantity of attacks over the last years, a trend that seems to continuously increase. This is why both academic researchers and companies are investing a large amount of money to secure and protect their networks.

This thesis gives its contribution to the literature by presenting an intrusion detection system that uses a number of different anomaly detection techniques to detect attacks against web servers and web based applications over the HTTP protocol. The system analyzes client queries that reference server side programs and creates models for a range of different features of these queries. Examples of such features are the length and the byte distribution of a certain parameter. In particular, the use of application specific modeling of the invocation parameters allows the system to perform focused analysis and produce a reduced number of false positives.

Contents

Chapter 1

Introduction

Attacks against web applications pose one of the most serious security threats to modern computer systems. Not surprisingly, an explosive growth in the amount of security incidents involving web applications has been observed in recent years. According to a report presented by the cyber-security company Imperva, the overall number of new vulnerabilities in 2018 (17,308) increased by 23% compared to 2017 (14,082) and by 162% compared to 2016 (6,615). More than half of web application vulnerabilities (54%) have a public exploit available to hackers. In addition, more than a third (38%) of web application vulnerabilities do not have an available solution, such as a software upgrade workaround or software patch [6].

Web application vulnerabilities create opportunities for hackers to launch devastating attacks. Sophisticated attack techniques have enabled hackers to launch large-scale attacks more quickly. Hackers have also become more organized, building criminal networks and sharing exploits in underground forums. New automated attack tools now leverage search engines to rapidly discover and attack tens of thousands of sites and, for even greater efficiency and scale, hackers have built networks of bots (remotely controlled devices) to unleash large-scale attacks.

The most popular cyber attacks are currently *injections* and *Cross-Site scripting* (XSS). The first type accounted for 19% (3,294) out of the total vulnerabilities of 2018, which is also a 267% increase from the previous year. When speaking about injections, the first thing that comes to mind is SQL injections, but also remote command execution (RCE) are becoming very common. On the other hand XSS vulnerabilities continued to grow steadily and appears to be the second most common vulnerability (14%) among 2018 web application vulnerabilities.

A security breach involves severe losses for a company. The global average cost of a data breach for the 2019 study is $3.92 million, a 1.5% increase from the 2018 study, as stated by a report from IBM in [7]. The average total cost of a data breach climbed from $3.5 million in 2014, showing a growth of 12% between 2014 and 2019.

There are many different solutions to protect the applications and the network of a company: firewalls (and in particular web application firewalls, WAF), intrusion detection systems (IDSs) and intrusion prevention systems (IPSs).

An intrusion detection system (IDS) monitors network traffic for suspicious activity and issues alerts when such activity is discovered. As a consequence, it does not alter the network packets in any way. There are several types of intrusion detection systems, which employ different techniques:

- **Network-based Intrusion Detection Systems (NIDS)**: it analyzes the inbound and outbound network traffic to and from all the hosts in the network.
- **Host intrusion detection systems (HIDS)**: it monitors only the traffic related to the host in which it runs. It is more precise than a NIDS and can identify malicious data generated by the host itself (for instance if infected).
- **Signature-based intrusion detection systems**: match the packets observed in the network against a database of signature of known attacks.

- **Anomaly-based intrusion detection systems**: it analyzes the network traffic and raises alerts if something is suspicious with respect to an established baseline.

Unlike an IDS, an intrusion prevention system is not limited to traffic monitoring only, but it also provide a control mechanism. While an IDS is usually passive in the sense that it logs the observed activity without performing any active action, an IPS deeply inspects the packets and can execute a real-time response to stop an immediate threat to the network. There are three common response actions:

- Terminate the TCP session that has been used to carry the attack and block the offending source IP address or user account from accessing any resource in the network.
- Reprogram or reconfigure an existing firewall to prevent a similar attack occurring in the future.
- Remove or replace any malicious content that remains on the network following an attack. This is done by repackaging payloads, removing header information and removing any infected attachments from file or email servers.

IDS and IPS have in common many features. They both suffers from the generation of false positives. However, while for IDS this is not a big issue (because no real action is taken and it is up to the system administrator to decide what to do), an IPS automatically takes countermeasures against the identified intrusions and it can lead to the denial of legitimate traffic. As a consequence this latter kind of systems requires very fine tuning of the parameters.

In this context lies HTTPServer, an intrusion detection system that performs anomaly detection on the HTTP traffic flowing across a network to secure. The goal is to identify and report attacks against web servers and web based applications. HTTPServer applies a custom machine learning algorithm to profile the HTTP traffic targeting a certain web application and to be able to detect outliers, which usually are symptoms of malicious inputs.

One key feature that distinguishes this anomaly detector from the other systems presented in literature is that HTTPServer performs *unsupervised* machine learning, instead of *supervised* machine learning.

In Supervised learning, the model is trained using data which is well "labeled." It means that some data is already tagged with the correct answer (for instance benign or malicious). Therefore, a supervised learning algorithm learns from labeled training data and helps in predicting outcomes for unforeseen data. On the contrary, unsupervised learning is a machine learning technique where it is not needed to supervise the model. Instead, it works on its own to discover information and it mainly deals with the unlabelled data.

HTTPServer is designed to process data in real time, as soon as they are received. As a consequence, supervised machine learning is not applicable. The learning phase for an unsupervised algorithm is usually more complex to implement because there is not a data set with both benign and malicious inputs that trains the model and allows it to understand what is good and what is bad. However, HTTPServer is able to automatically adapt itself to the traffic targeting the specific web server it is protecting and it is robust to the presence of outliers.

HTTPServer is a module of LLAnta, the company's network analysis tools. In the following sections I will give a description of the company an its products, in order to provide the context needed to understand how HTTPServer works and it has been developed. Next there will be a review of the work done in literature and from which I took inspiration. It follows a detailed explanation of the system structure and how the different components interact together. Finally, I will show the results and present my final consideration and what I plan to do as future work.

Chapter 2

Lastline

This chapter contains an introduction to the enterprise where I spent my internship period, *Lastline, Inc.*. Over the years, Lastline has developed a full ecosystem of components that are involved in processing network data and produce outputs. A general understanding of the interactions among the various elements is required to achieve a better comprehension on how the system I built works and it is integrated inside the already established ecosystem.

2.1 The Company

Lastline, Inc. is an American cybersecurity company founded in 2011 by Engin Kirda, Christopher Kruegel and Giovanni Vigna and based in Redwood City, California. Its first objective is to protect enterprise's networks from the increasing number of threats in the digital world, allowing them to stop intrusions in advance and prevent possible dangerous data breaches.

Lastline, Inc safeguards every aspect of a company network and focuses on six main security challenges [13]:

- **Protect Public Cloud Workloads**: secure both the internal and external public cloud traffic in *infrastructure-as-a-service* (*IaaS*) environments, like the well known *Amazon Web Services* (*AWS*).
- **Accelerate Threat Response**: quickly cut through the noise, understand the most urgent threats and drive the correct response, avoiding to detect the intrusion when it is too late.
- **Detect Lateral Movement**: identify the anomalous behavior of the internal compromised systems, as the attack spreads inside the network.
- **Block Unauthorized Access**: recognize unauthorized access from inside (for instance because of an attacker using stolen credentials) or outside the customer network.
- **Prevent Data Exfiltration**: prevent leakages of confidential data.
- **Secure Any Email System**: additional layer of defense for cloud email as well as customer-managed email systems. It protects users from advanced email security threats that are engineered to defeat other security tools, such as *spear-phishing*, *ransomware*, *credential stealers*, and other malicious emails.

Lastline, Inc. provides an innovative and unique solution to face all these hazards by using Artificial Intelligence (AI) as the main engine of the system. Artificial Intelligence describes the ability of computer systems to simulate intelligent human behavior. This often includes capabilities such as learning information from the processed data, taking decisions and being able to automatically correct an erroneous behavior.

The majority of the other existing solutions applies the AI to network (and user) behaviors only in order to find anomalous patterns of behavior within the network traffic. However, by looking solely at anomalies, one would incur the risk of being flooded by anomalous-yet-benign events (i.e. false positives), which are commonplace in most networks. At the same time, one would be blind to malicious events that do not generate any anomaly.

Reducing the number of false positives is a task of tremendous importance for this kind of application, since the quantity of processed data is very large. Consider, for instance, a network in which, every day, a server receives 1 million *HTTP* requests. A false positive rate of 0.1%, which seems very low, would generate 10000 alerts per day, which would be likely not easily manageable for most organizations.

Lastline, Inc. copes with this issue by training AI automatically, both on network traffic and malicious behaviors. This unique combination enables deterministic detections and eliminates most of false positives. In addition, unlike the competitors, the company performs deep inspection on the analyzed traffic, extracting not only high-level information but events at several different abstraction levels (from raw packets to network flows). This allows for richer input data and the creation of more precise models.

2.2 The Lastline ecosystem

Lastline has built a scalable and distributed architecture that allows a Managed Security Services Provider (MSSP) to deliver next-generation managed security services to protect its customers against advanced threats.

An MSSP provides outsourced monitoring and management of security devices and systems [8]. The main network security services include firewall, intrusion detection and threat intelligence.

As illustrated in figure 2.1, Lastline's products can be integrated in the tools offered by the MSSPs. On top of Lastline technology, they can easily add further value for their customers, as the platform allows MSSPs to add their own tools.

Lastline's architecture for MSSPs is made of four modules:

- **Sensor**: inspects inbound and outbound traffic to protect the network from web threats and malicious artifacts (binaries, documents, and email attachments) entering the customer's networks. It also forwards unknown objects and the alerts related to malicious connections to the Manager for additional analysis.
- **Engine**: is the high-resolution sandbox based on full-system emulation. Engines receive the artifacts from the Manager and analyze them, detecting advanced malware. The results of the analysis are shown to the customers in the Manager's dashboard.
- **Data Node**: analyzes network data to identify anomalous activity on the network.
- **Manager**: is the core component of the architecture that correlates all data coming from Sensors, Data Nodes, and Engines. It configures the Sensors, it provides the alert dashboard and reporting interface and it mediates the communication between the Sensors and the Engines.

During my internship period I mainly worked on the Sensors and the Data Nodes (with the LLAnta project). Therefore, the following two sections provide a more detailed description of these components.

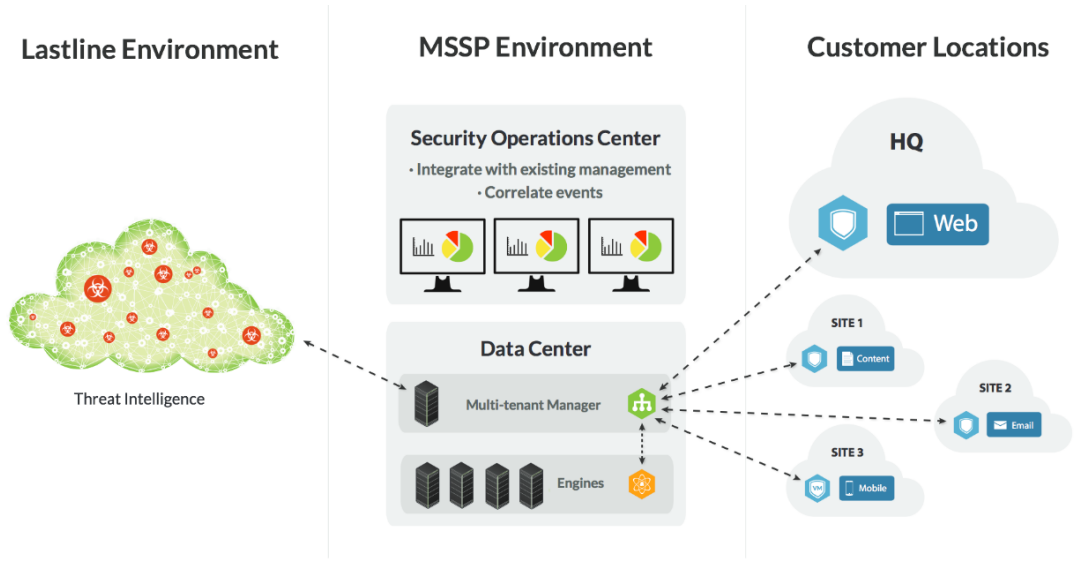

Figure 2.1: MSSP Deployment Model

2.3 The Sensor

The sensor represents the front line of defense for all the Lastline's products, upon which all the detection mechanisms are built. It is the core component that inspects the network traffic entering and exiting the customer network and performs all the operations needed to guarantee the security of the monitored environment.

Essentially, a sensor is a little box running a modified version of the Ubuntu Xenial (16.06) operating system that contains a number of proprietary components used to perform the analysis on the network traffic and to communicate with the upstream Manager.

A sensor can be deployed in various operating modes to accomplish different degrees of protection. A more detailed description of these configurations and the related components interactions is provided below.

2.3.1 Simple Sniffing

The sensor architecture in this configuration is shown in figure 2.2. The data are extracted out of the network card by two sniffing components:

- **llpsv**: it performs fast and lightweight processing on packets. Mainly it matches Lastline's blacklist for hosts and IP addresses, it parses DNS traffic and generates netflow logs.
- **suricata**: extension with proprietary patches of the open source version of Suricata, a well-known intrusion detection system (IDS). It performs full deep packet inspection, applies Lastline's network signatures, extract files and extracts various types of application layer protocol logs (such as HTTP, Kerberos and SMB).

Even though it may seem redundant to have two different sniffing components, llpsv and suricata perform completely different jobs. As delineated by figure 2.2, the output of llpsv reaches directly the **llshed** component, while the output of suricata follows a more complex path.

The main goal of llshed is to ensure that all the inputs it receives from the other sensor components are successfully uploaded to the backend (i.e. the Manager). In case of a temporary communication failure, llshed will continue to retry the upload of the information until it succeeds. This safeguards customers from data loss even in case of temporary connection issues with the Manager.

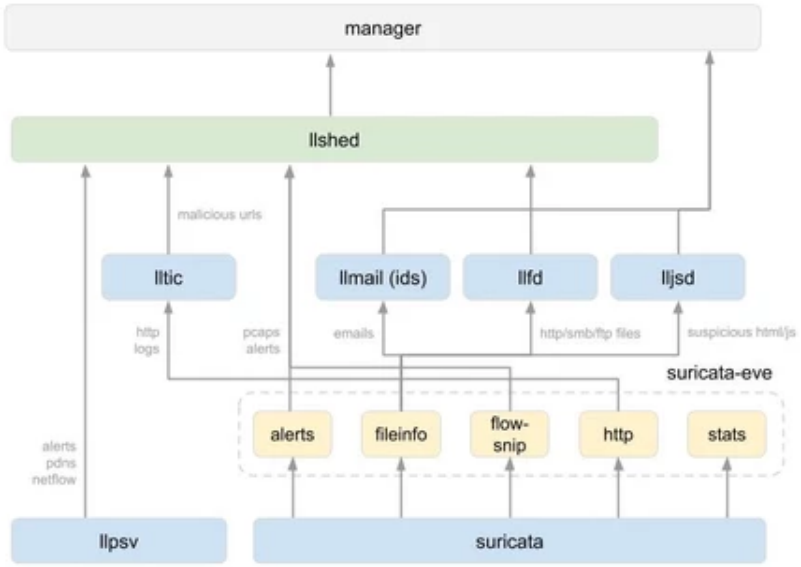

Figure 2.2: The Sensor components in simple sniffing mode.

Moreover, llshed is in charge of enforcing several types of filtering policies and aggregation for the data generated by the sensor, in order to avoid to upload to the backend too much information. This is accomplished by bucketing the information and giving priority to interactions with rarely contacted endpoints or endpoints with unknown or low reputation.

For instance, connections to domains such as *facebook.com* or *twitter.com* are quite common and likely to be benign. Therefore, if the sensor captures many connection to these domains and maybe one connection only to another unusual domain, it will give priority to the latter. As a result, even if not all the information is provided to the manager, it is unlikely to miss relevant (i.e. malicious) traffic.

As mentioned above, llpsv produces artifacts ready to be sent to the Manager, while the output of the suricata component needs to be processed by another set of workers, that constitute the so called **suricata-eve** element. There are currently five workers in charge of different jobs:

- **alerts**: IDS alerts generated by suricata are first converted into the format used by Lastline and then sent to llshed for upload.
- **fileinfo**: any file extracted from the traffic requires additional analysis before it can be uploaded. There are currently three different cases:
  - **Emails**: email data are processed by the **llmail** component.
  - **Interesting files**: interesting file observed over the HTTP/FTP and SMB2 protocols are collected. A given file is considered relevant if it matches a set of custom rules defined by the company. These files are submitted to the **llfd** service, that applies a pre-filter to decide whether the file should be uploaded to the Manager for further inspection.
  - **Suspicious HTML and JavaScript**: a special collection of rules is used to flag HTML or JavaScript code that is likely to contain malicious content. The collected data are sent to the **lljsd** daemon, that applies heuristics on each extracted document. If it is considered suspicious, its entire content will be uploaded to the backend.
    This worker mainly addresses malicious web pages containing *drive-by downloads* attacks.
- **flow-snip**: Lastline's proprietary version of Suricata is able to build a small pcap containing packets of the flow that produced a specific IDS alert. These files are then sent to llshed for upload to the backend.

- **http**: suricata generates a log for every HTTP message sent to or from the customer network. These logs are both uploaded to the Manager and submitted to the **lltic** component. lltic is the Lastline's threat intelligence cache used for local detection. If any of the URLs seen in the logs is considered malicious, lltic has the capability of generating an alert.
- **stats**: Suricata collects detailed statistics on the operation of its different components on a regular basis. These stats are uploaded to the Manager.

2.3.2 ICAP

ICAP (Internet Content Adaptation Protocol) is a lightweight protocol designed to off-load specific content to dedicated servers, thus freeing up resources and standardizing the way in which features are implemented. It was born from the need to ease the number of value-added services the web servers have to provide and to reduce the overhead in providing these services to the customers.

ICAP is mainly used as a vector for HTTP services, although the communication that it can handle is not restricted just to this protocol. An ICAP client has two basic operations modes:

- **Request modification** (REQMOD): in this configuration, the ICAP client will relay to the ICAP server all the incoming HTTP requests before relaying them to the actual origin server. The ICAP server can inspect the content of the HTTP request and it can even make some modifications.
  This mode is used to provide cached content, to redirect an unauthorized/restricted request to another page (content filtering) or to prevent clients from exfiltrating data towards low reputation domains.
- **Response modification** (RESPMOD): in this scenario, the ICAP client will share with the ICAP server the HTTP response generated by the origin server before delivering it back to the client. In this way the ICAP server has the capability to see the server response and, for instance, it can perform on-the-fly virus checks and block the clients from downloading malicious documents.

The basic concepts of the protocol are illustrated in figure 2.3.

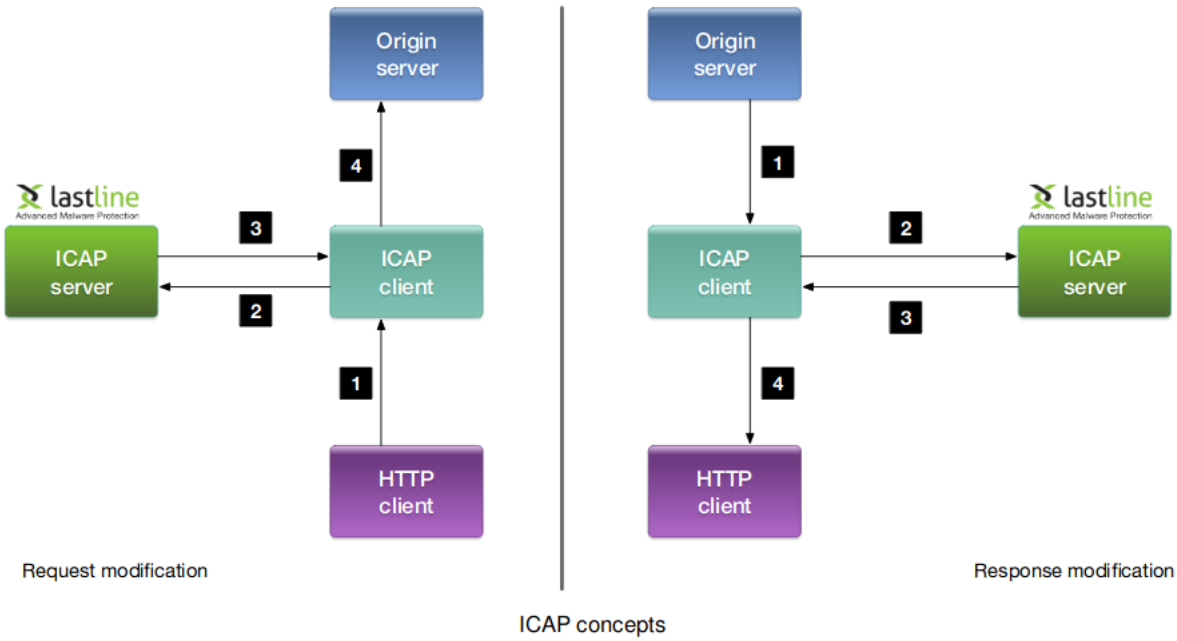

Figure 2.3: ICAP basic concepts.

Commonly, there is an ICAP server for each dedicated function (virus scanning, content filtering...), in order to provide a proper standardization.

The Lastline appliances can handle both REQMOD and RESPMOD requests at the same time, offering the maximum level of protection and, if the client implements it, they support the use of the ICAP Preview functionality. In this scenario, rather than sending to the ICAP server the entire HTTP transaction, a client can start by delivering the beginning of such transaction and let the ICAP server itself decide whether the transaction should be skipped or fully delivered. The Lastline ICAP implementation has the capability to derive from the preview content the file type of the document being served, and therefore determine whether it is potentially malicious.

Nowadays, ICAP is used to extend transparent proxy servers. A proxy is a server that sits between a client and a content provider (i.e. a web server). As with the ICAP protocol, when a client sends a web request to a certain server, the proxy intercepts the request and performs some processing before forwarding it to the server. The term transparent is used to distinguish a transparent proxy from an ordinary one. Contrary to traditional proxies, it is invisible to user (i.e. it does not require any client side configuration) and it is not allowed to modify the request sent by the client. Transparent proxies have three main applications:

- **Proxy caches**: store the content requested by the client. Therefore, if a client requests the same content, the proxy is able to provide it itself, reducing the load on the web server. This also allows to save bandwidth and reduce loading times. In order to avoid consuming too much space, the cache entry related to a certain content usually expires if no client requests the same content in a given amount of time.
- **Filtering proxies**: prevent access to certain websites or web services. This is usually done by companies to restrict the services their employees can access to. It is also used to monitor users activity.
- **Authenticate users**: this is frequently implemented by companies offering a public WiFi to let the users use Internet after they agree to their terms and conditions.

Lastline's sensors can be configured to operate as ICAP servers to provide additional security to their customers. However, if the customers use the HTTPS protocol instead of HTTP to perform their web requests, all the traffic is encrypted and ICAP becomes useless.

This is why, on top of the ICAP configuration, a sensor can work as an explicit proxy. In such a case, the sensor is able to perform TLS decapsulation on the HTTPS traffic(i.e. it is able to decrypt the proxied traffic and process the content in clear text).

The diagram in figure 2.4 represents the information flow of an explicit proxy sensor. Beside the detection capabilities provided in Simple Sniffing mode, it is possible to generate the following events:

- **events on blacklist hits**: whenever the user attempts to visit a low reputation domain, ICAP will prevent access. The reputation information is obtained by ICAP by making requests to the lltic service (described above) on each visited URL.
- **events on malicious file downloads**: whenever a file is detected by ICAP, it is shared with the llfd service for analysis. ICAP then periodically checks for the analysis progress until a score for the file has been determined by llfd.
- **events on suricata IDS detections**: if Suricata detects something anomalous, an entry related to the suspicious content will be added to the lltic cache and blocked by ICAP as if it was a blacklist event. This scenario becomes possible only if the sensor works as an explicit proxy. In fact, in this situation, Suricata is able to monitor the unencrypted traffic flowing across the sensor.

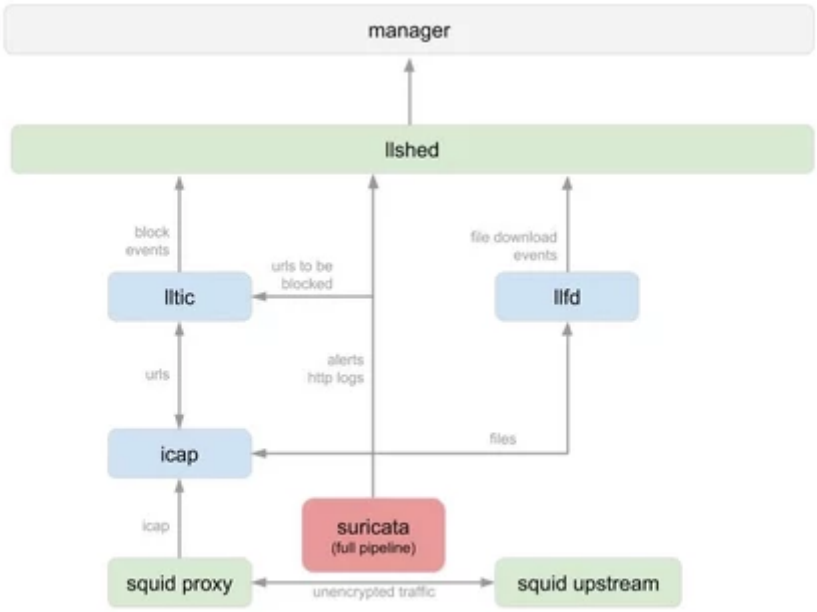

Figure 2.4: The Sensor components in Explicit proxy/ICAP mode.

To sum up, the complete sensor data and process flow are shown in diagram 2.5.

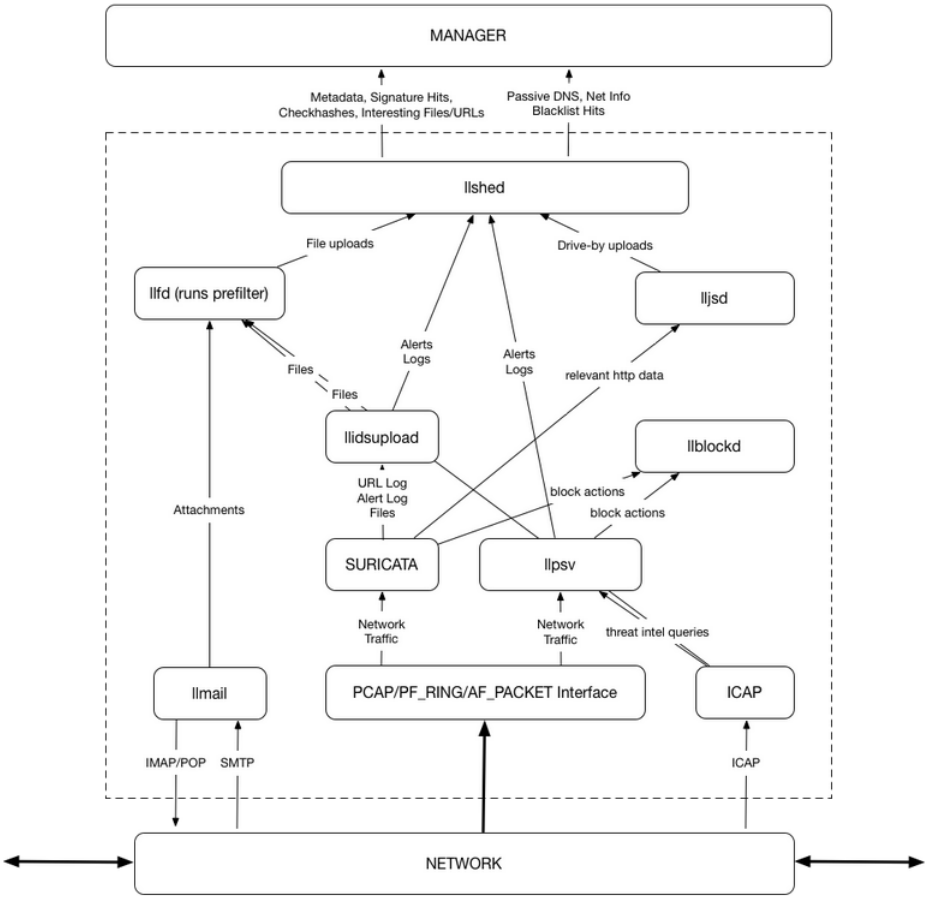

Figure 2.5: The Sensor data and process flow.

2.4 LLAnta

LLAnta (LastLine Advanced Network Traffic Analysis) is a network traffic analysis system that runs inside the Data Nodes and communicates with the Manager. It receives inputs from multiple sources, such as Lastline's sensors, other network firewalls or the monitored hosts themselves, and processes them to create models (statistical profiles of hosts inside the network).

LLAnta analyzes the data it receives using a combination of machine learning, statistical analysis and content analysis. It is able to automatically detect anomalous activity in a monitored network and supports hunting for suspicious activity.

Currently, three kinds of inputs are managed:

- Netflow: netflow data.
- PDNS: DNS data.
- URL: rich information on each observed URL.

After the elaboration of the input information, *LLAnta* can produce two different outputs:

- **Facts**: properties associated to hosts inside a monitored network. An example of a fact is the operating system running on a host, or the host role inside the network (such as name server or email server).
- **Alerts**: caused by anomalies identified analyzing facts, possibly with respect to the models. An example of alert is represented by anomalous traffic types (for instance a nameserver that suddenly generates HTTP traffic).

These outputs are sent to the Manager for further processing and are shown to the customers in the UI (User Interface).

2.4.1 Architecture

The general architecture is shown in figure 2.6.

The LLAnta service communicates with the external environment (i.e. the Manager) thanks to the *llupload worker*. This takes care of fetching the data from the Manager and storing them in the appropriate queue, one for each supported input. For each queue there is a corresponding processor element (NetflowProcessor, URLProcessor and PassiveDNSProcessor) that is in charge of retrieving the input from its queue and sending it to the service.

The LLAnta service and the worker exchange information using a bidirectional protocol (request-response protocol). For every request, the client should always expect a response, that can potentially be empty.

There are four possible types of request the llupload worker can send to the LLAnta service:

- **ProcessRequest**: used to submit new data to the service. The data type (netflow, URL, PDNS) defines how the data field should be processed.
  The response to this request is empty.
- **FlushRequest**: the flush operation is a periodic operation that triggers the collection of the facts and alerts produced by the service. The output data are sent to the llupload worker, where the FlushProcessor component stores them in a queue. These data will then be sent to the Manager for further processing.
- **ConfigureRequest**: used to configure the service with specific settings.
  The response to this request is empty.
- **QueryStatusRequest**: query the service and retrieve status information.

The llanta services manages multiple instances of LLAnta using a master-slave model. The master (*Muxer master*) is in charge of routing the requests received by the *llupload worker* to the various slaves (*Muxer slaves*). Each slave runs in a separate process and can handle multiple LLAnta instances thanks to the *BasicMuxer* component.

An example of possible workflow is the following:

1. A collection of LLAnta inputs, for instance coming from a never-seen-before sensor, is received by the LLAnta service, encapsulated in a ProcessRequest message.
2. The request is handled by the Muxer master and there are three possible outcomes:
   - If a new slave can be allocated (i.e. the maximum number of possible running slaves has not been reached yet), a new process is spawned.
   - If it is impossible to create a new slave, the less-loaded one is chosen.
   - If all the slaves are fully loaded, an error message is logged and the message is discarded
3. On the slave, a new LLAnta instance is instantiated and the processing starts.
4. Each following ProcessRequest with inputs from the same sensor will be routed to the same slave, which will process the inputs in the same LLAnta instance.

2.4.2 LLAnta instance

A LLAnta instance processes all the received inputs (netflows, URLs, PDNS) and returns LLAnta *Facts* and *Alerts*. It is made of four main components: Network State, Plugins, Detectors and Context. The overall structure is shown in figure 2.7.

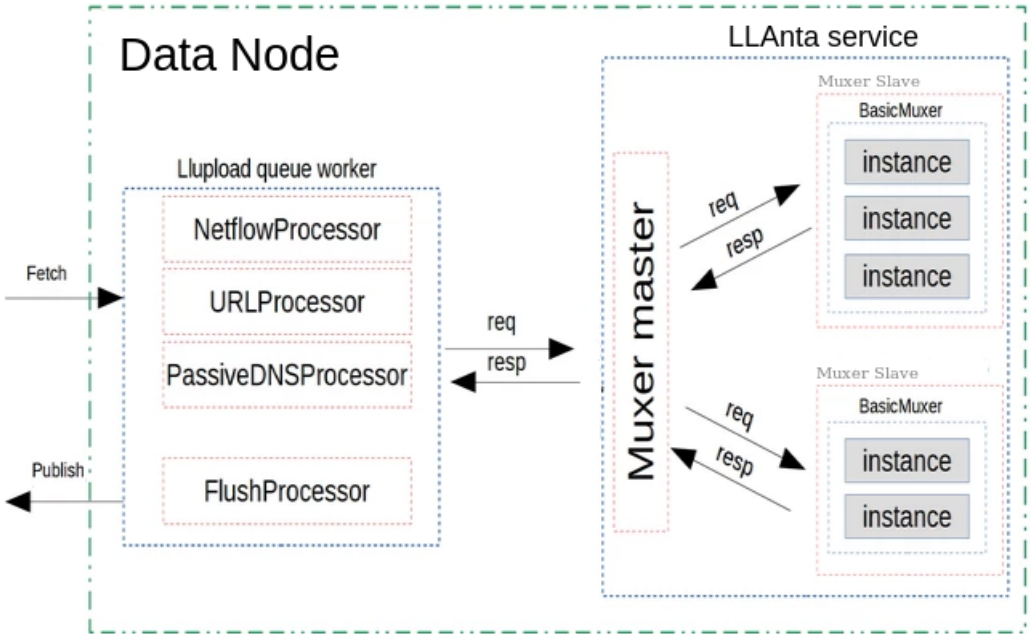

Figure 2.6: LLAnta architecture.

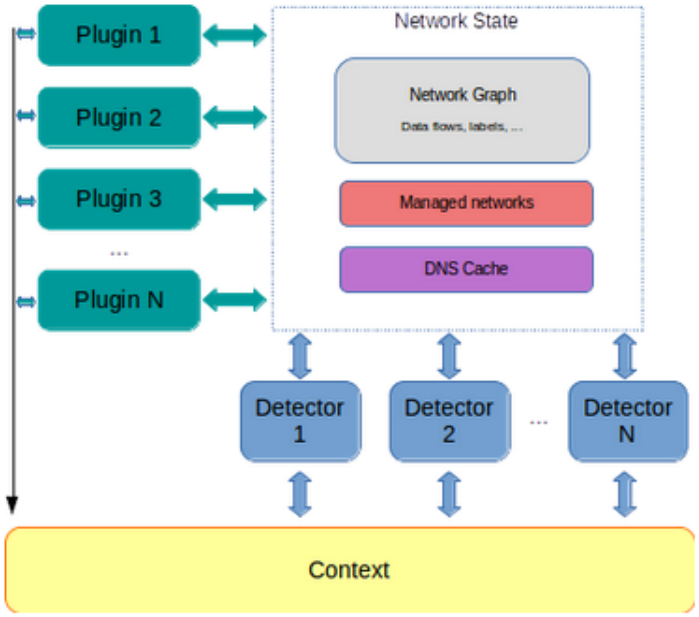

Figure 2.7: LLAnta instance.

Network State

The Network State is a model of the monitored network and it is comprises three elements: the Network Graph, the Managed Networks and the DNS Cache.

The first one is a directed graph modeling all the network interactions observed in the network. Each node of the composed graph is an IP address and an edge between two nodes exists if there has been a connection between the two IP addresses. The direction of the edge is determined according to the source and destination of the connection.

The data flows represented by the edges are enriched with a variety of metadata merging information obtained from different data sources. For instance, a web request between two hosts will create a data flow between the two endpoints and include high level HTTP information about the request. Moreover each entry in the graph has an expiration timeout. When a new node or edge is created, it is assigned with the timestamp of the input source that generated that entry. The timestamp is updated each time a new input is received with the same data. The network periodically checks its entries and delete every element whose timestamp is too old (according to a configuration parameter).

With the Managed Networks, instead, LLAnta tries to define the boundaries of the customer network based on the analysed traffic. A Managed Network consists of a set of IP addressed that belongs to the same CIDR (Classless Inter-Domain Routing). A customer network can have multiple networks configured, with separated IP ranges over different CIDR blocks. This abstraction is extremely important to LLAnta to be able to apply its logic. It is in fact needed to distinguish between hosts internal and external to the customer network and to properly correlate the event seen in the traffic.

Context

The LLAnta context is the main way to exchange unstructured information between plugins and detectors. It can store anything that a plugin/detector wish to store, in opposition to the Network State, that can only contain specific information (for instance the DNS cache only holds associations between an IP address and the corresponding domain name).

The context consists of a key-value structure which is periodically serialized on disk for resiliency. This is useful for two main reasons:

- In case of failure of a LLAnta instance, its state can be restored from the disk and the loss of data is minimized.
- The context state can be populated offline, for instance with data collected in the past, and then it can be used to test the behaviour of plugins and detectors.

Like in the Network State for the Network Graph, an entry in the context is deleted after a certain amount of time has elapsed without the information being refreshed.

Plugin

Plugins are components that operate simple and fast actions on every single input entering the LLAnta instance (netflows, URLs, PDNS). In order to receive an input of a certain type, a plugin must register itself as processor of that input family.

A plugin performs two main jobs:

- Generate Facts about a host: plugins can output Facts learned from the inputs.
- Populate the state for the detectors: a plugin can be considered as a an aggregation stage for the data that will be analyzed by a detector. It can store information in the Network State and in the Context. When a detector is scheduled, it can access all these data and get all the information needed to be able to decide if it is needed to generate an alert or not.

LLAnta counts more than twenty different plugins, each one designed for a specific purpose, such as port scan and DNS tunnelling detections.

Detector

Detectors are modules that work in cooperation with the plugins. Similarly to the plugins, detectors need to register themselves to the input family they want to process, but they can perform heavier computation, because they are not scheduled any time a new input is received. Moreover, while the plugins can only generate Facts, detectors are in charge of generating anomaly alerts.

A LLAnta Alert is generated for a specific host when the logic of the detector matches with data read from the Network State and/or the Context.

2.4.3 LLAnta Manager

Each LLAnta instance is has a LLAnta Manager component, that controls the scheduling of plugins and detectors. Respectively, Plugins are handled by a *PluginManager*, while a *DetectorManager* takes care of the detectors.

In general, while plugins are scheduled at every ProcessRequest message sent by the llupload worker, the DetectorManager, in presence of a ProcessRequest request, runs a detector only if:

1. The detector is not already scheduled for the given host.
2. The TRIGGER_TYPE of the detector corresponds to the type of the item to process.
3. The trigger_check of the detector is satisfied for that item (i.e. the not only the input data is correct, but there are also enough data to actually perform the required computation).
4. An amount of time defined by the SCHEDULE_DELAY parameter has elapsed since the detector was scheduled. This is often needed because, if a detector is run too often, it may not acquire enough information to actually generate the alerts.

The (high level) class diagram in 2.8 shows the architecture of the classes, and 2.9 summarizes the detector scheduling workflow.

2.4.4 LLAnta batch processing

All the mechanism described above happens in *streaming*, i.e. the inputs are processed as soon as they are received and the results are given straightaway.

LLAnta supports another working mode, the batch (or offline) processing workflow. In this scenario, batch jobs can load the Network State and the Context of a specific LLAnta instance from stable storage, they can perform the required analysis and output both Facts and Alerts (along with the ones produced by plugins and detectors).

This operation mode results particularly useful for jobs that do not need real time data and/or are computationally heavy to run (therefore they can affect the performance of the whole instance). Moreover, it can be used to test offline a certain algorithm before deploying it to work in real time inside plugins or detectors.

LLAnta jobs run periodically on Data nodes, with a scheduling granularity that can be customized for the single job. In this way different batch jobs can be scheduled to run periodically with different intervals.

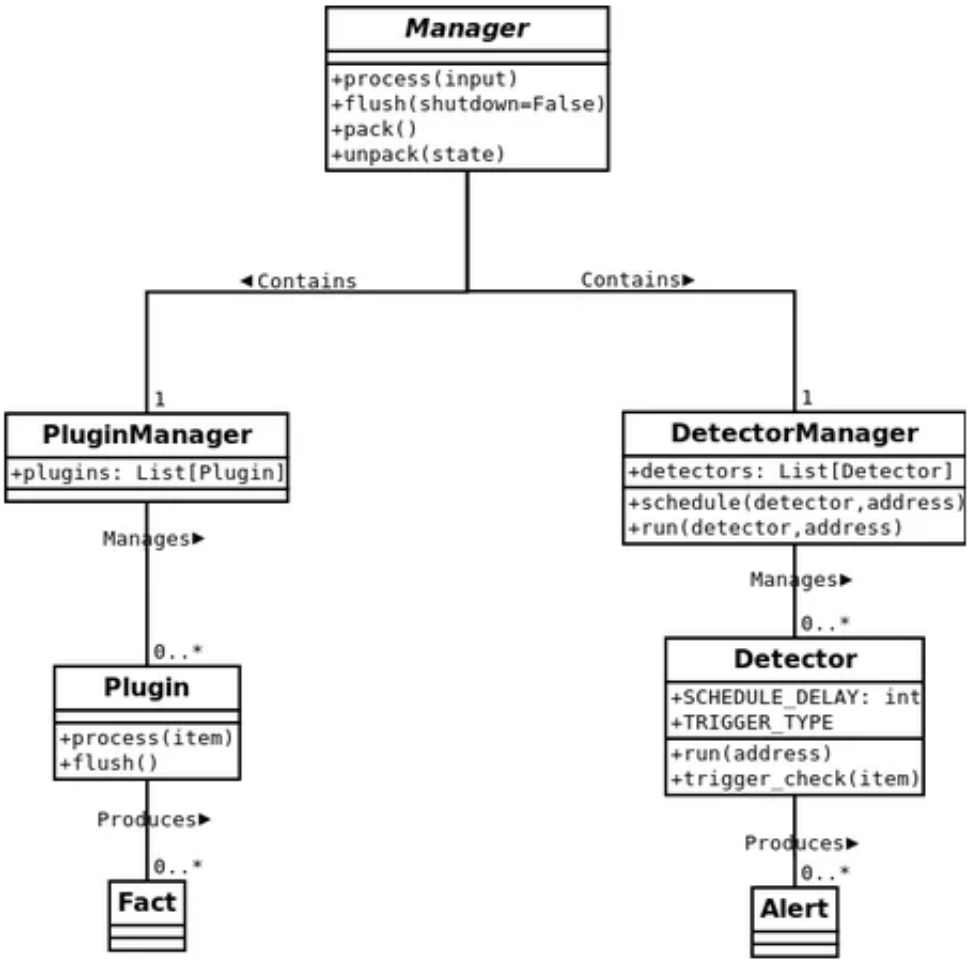

Figure 2.8: High Level class diagram of a LLAnta instance.

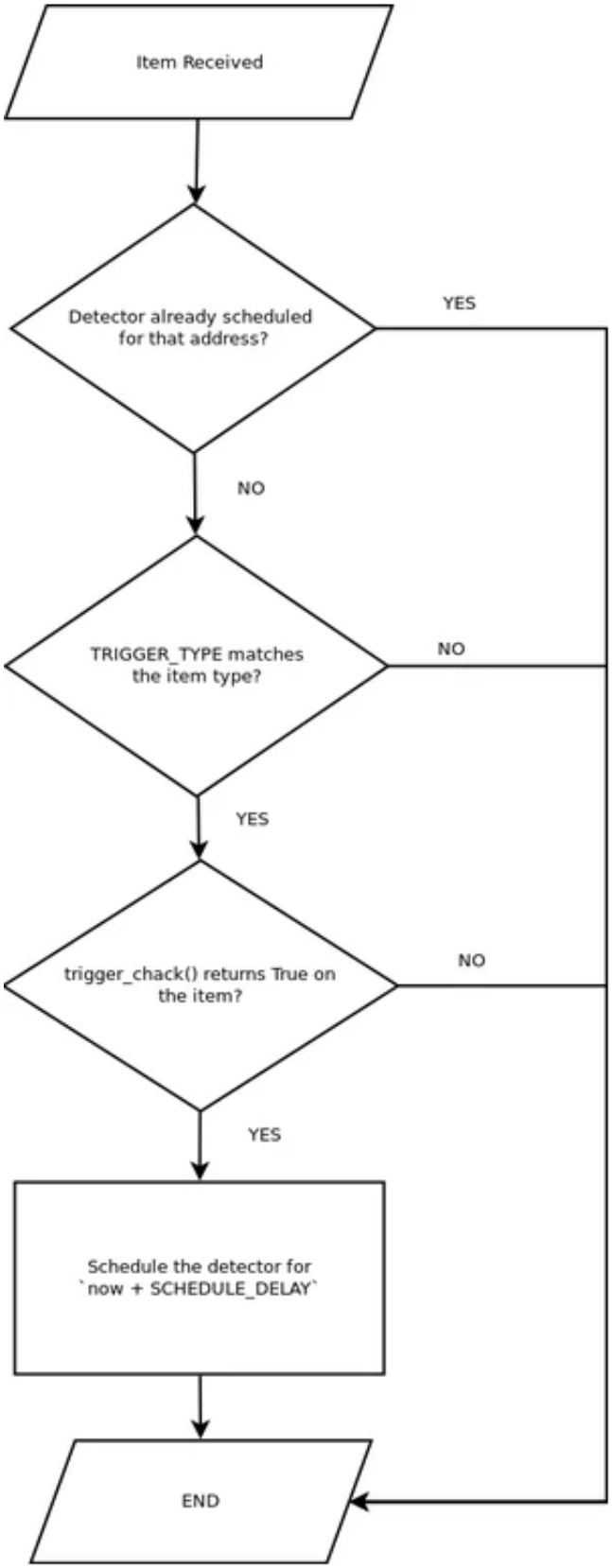

Figure 2.9: Flow chart of the detector scheduling process.

Chapter 3

State Of The Arts

In this chapter I provide an introduction to the basic knowledge that will allow even a non-expert reader to understand the development of this master thesis. This includes a brief description of the network protocol on which my job focuses, the HTTP protocol.

It follows an analysis of the work done in literature, with a particular attention to one solution from which I took inspiration, elaborating and improving the idea it as much as possible.

3.1 The HTTP Protocol

HTTP (Hypertext Transfer Protocol) is an application layer protocol which runs over the *TCP* transport layer protocol. The first version was designed in the early 1990s and it has evolved over time. Nowadays it exists also a secure version of this protocol (HTTPS) that is sent over a *TLS* encrypted connection.

HTTP is used to exchange *resources* all over the World Wide Web. The most common type of *resource* is a file, but it can be anything that it is identified by an URL (Uniform Resource Locator).

An URL, commonly called web address, is a sequence of five hierarchical components (as shown in figure 3.1):

1. A **scheme** ("http" for the HTTP protocol) followed by ":".
2. An optional **authority**, made of three subcomponents:
   - An optional **user info** which provides the username and the password of the user (deprecated for security reasons).
   - An **host**, which can be an hostname or an IP address.
   - An optional **port** number.
3. A slash-separated **path** defining the resource to retrieve.
4. An optional **query** used to send parameters to the server.
5. An optional **fragment** that provides direction to a secondary resource, such as a section heading in an article identified by the rest of the URL.

HTTP is a client-server protocol. An entity, the *user-agent*, sends an HTTP request message to the server, asking for a *resource*. The server handles the request and provides an answer sending an HTTP response. The *user-agent* can be any tool that acts on behalf of the user, but most of the times is a regular web browser. This process is represented in figure 3.2.

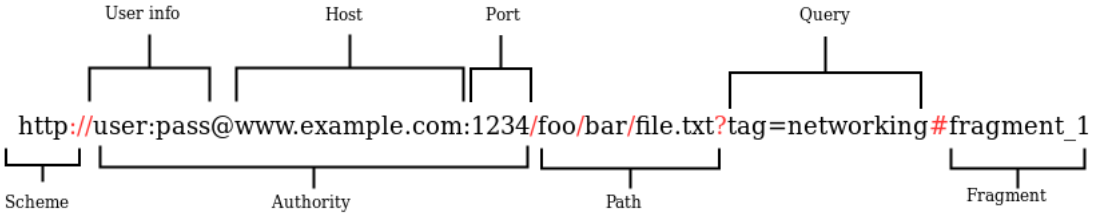

Figure 3.1: Example of an URL including all the optional components.

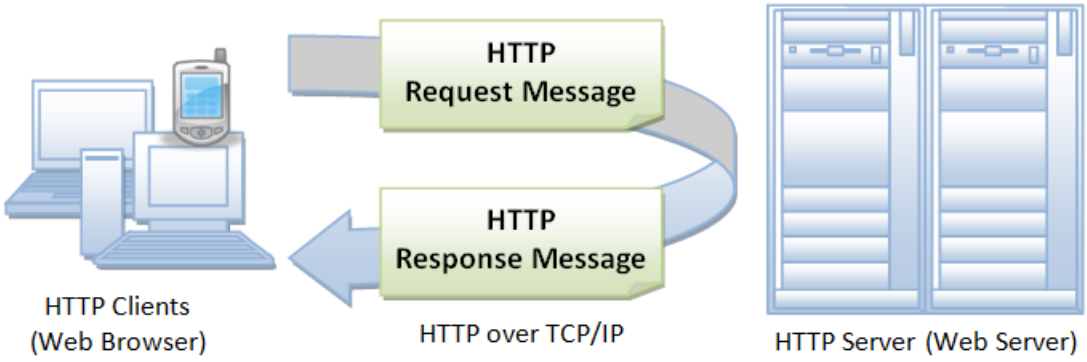

Figure 3.2: The HTTP request/response pattern.

HTTP is a stateless protocol, which means that each HTTP requests is self-contained and independent from any other request a client sent before. However, this can be problematic for web servers that want to answer coherently to several inputs from the same client. Thus, this shortcoming has been addressed by taking advantage of the HTTP extensibility and introducing HTTP cookies. An HTTP cookie (also called web cookie or browser cookie) is a small piece of data that a server sends to the user's web browser. The browser may store it and send it back with the next request to the same server. It remembers stateful information for the stateless HTTP protocol and it is typically used to create the concept of session between a client and a server.

HTTP is generally designed to be simple and human readable. The request and response messages have a well defined structure and they can be read and understood by humans, which is very helpful for debugging purposes.

An HTTP request message is composed by:

- A request line. This contains the HTTP method that defines the operation the client wants to perform, the path of the resource addressed by the request and the version of the HTTP protocol in use. Most of the times a client wants to retrieve a resource (using the GET method) or send values to the web server, such as a form (using the POST method).
- Optional HTTP headers used to provide the web server with additional information.
- An empty line.
- An optional message body, used by some methods (such as the POST method) to send data to the web server.

Accordingly an HTTP response message consists of:

- a status line including the version of the HTTP protocol in use, the status code and the status message, that indicate the outcome of the corresponding request. For instance, for a successful HTTP request the status code is 200 and the status message is "OK".
- HTTP response headers like for the HTTP request.

- An empty line.
- An optional message body containing the fetched resource.

This is a very short overview of the HTTP protocol that enables a non-expert reader to understand the basics of how resources are exchanged over the web. HTTP deals with many other aspects such as access control, authentication and caching, but these topics are not addressed here and they are left to the readers to be explored.

3.2 Literature Review

Web-based vulnerabilities have been of great interest in the academic world over the last years. Attacks against web servers are one of the most serious security threats to modern computer systems and this trend does not seem to be likely to change soon. Therefore, many researchers have coped with this problem and they came out with several different solutions, each of them with both drawbacks and advantages.

One very popular approach is to develop a *misuse detection* system. This usually comes in the form of custom signatures to detect signs of web-based attacks. However, writing signatures is a very time consuming and error-prone process that entails some major disadvantages. First of all, they require a profound knowledge of all the possible hazards. Moreover, the signature database needs to be constantly updated to keep up with the increasing number of new vulnerabilities being discovered. Finally, the main problem of this kind of solution is that they can only detect known attacks, but they are completely harmless against the so called *zero-day exploits*. These are attacks that leverage on yet unknown vulnerabilities, therefore no signature can be written for them.

Another possible solution, instead, is to build an *anomaly detection* system. The goal is to try to profile the behavior of the web server to protect and report as anomalous every transaction that deviates significantly from the established profile. In this way, the detection capabilities are tailored for each web server and they do not rely on any known pattern. However, this approach is possible only under the assumption that the attack pattern is different from the normal behavior.

My work is related to this second method and, in particular, it takes inspiration mainly from the ideas presented by Christopher Kruegel, Giovanni Vigna, William Robertson in [1] and Tammo Krueger, Christian Gehl, Konrad Rieck, Pavel Laskov in [2]. In the following sections I will describe more in details these two papers, focusing on the challenges that need to be faced to improve what has been done.

3.2.1 A multi model-approach to the detection of web-based attack

Kruegel and al. propose an anomaly detection system following a *learning-based* approach. Their analysis is targeted to the identification of attacks in the parameters contained in the query string of an HTTP request. To achieve this goal, they rely on a training data set to build profiles of the normal, benign behavior of users and applications [1].

The detection process makes use of several different *models* that are applied to the query at various levels of granularity (some are related to a single query attribute, while other can take into account the whole query or even multiple queries at once). A model is a set of procedures used to evaluate a certain feature of a query [1]. By making use of multiple models at the same time, it is possible to increase the accuracy of the overall system and reduce the false positives.

The models are created for each specific program run by the web server, identified by the resource path in the HTTP request. This means that both the modeling and the detection mechanism are performed separately for each resource path provided by the web server.

Each model assigns an *anomaly score* to its related feature (either a single query's attribute of a whole query). This score represents the likelihood that a certain feature value occurs in the HTTP requests sent towards a given web server. The idea is that, if a given feature value has a

very low probability (i.e. an high anomaly score), there is a big enough confidence that the value under analysis is not benign. In other words, the goal of this system is to detect the *outliers* in the HTTP traffic, based on the assumption that benign requests have common feature values, while an attacker will probably send really rare content.

The system is organized in three stages:

1. First each model operates in *training* mode. During this initial training phase, the models profile the traffic they receive and learn which is the usual user behavior.
2. Then each model goes through a *validation* phase. After having gained enough information to determine the characteristics of normal events, the models are matched against a second set of HTTP requests. For each feature, they compute and store the highest anomaly score. Then they set up the threshold as a certain percentage higher than the computed maximum score.
3. After the validation phase, the models start operating in *testing* mode. Each model analyzes the HTTP traffic, computes the corresponding anomaly scores and, if a feature value has a score higher than the defined threshold, the query is reported as anomalous. Every model has the same weight regardless of the granularity level at which it operates, therefore it is sufficient that one model only reports an anomaly to flag the whole request a malicious.

However, a strong assumption is required in order for the system to operate as intended. The whole process requires that both the sets of HTTP requests analyzed during the training phase and the validation phase (respectively the training and the validation set) are clean from any attack.

This is impractical in real world environments for several reasons. First, it is very uncommon that regular HTTP traffic towards a publicly exposed web server is all benign because Internet is a public network and there is plenty of malicious sources. For instance, it is true that the anomalous data is often generated by an actual attack but it can originate also from more benign tools. In fact, there are many *vulnerability scanners* on the Internet that send inputs particularly crafted to resemble an attack for either research or testing purposes. However, this kind of input cannot be considered a real intrusion because there is no intention in exploiting a potential discovered flaw.

As a consequence, in order to build and validate the models, the system needs to collect in advance sample data of the HTTP traffic targeting the web server to protect and remove the anomalous feature values. This pre-processing phase adds a level of complexity to the whole system and has to be done manually by a system administrator.

Moreover this approach implies that training and validation are done *offline*, with data collected in the past, that can soon became obsolete. Therefore, It is likely that the models need to be periodically re-trained because the average user behavior can change over time and, if the same models are kept for too long, it is possible that they would start flagging as anomalous feature values that instead are perfectly normal. This would require to run the pre-processing phase every time the models need to be re-trained and validated, which can be very expensive.

Kruegel and al. evaluated the performance of the system using three different sample data sets (a web server located at *Google, Inc*, one at the *University of California* in Santa Barbara and on at the *Technical University* in Vienna). The main goal is to be able to detect the anomalies while keeping as low as possible the false positives. In their experiments, they were able to reach a false positive rate between 0.01% and 0.07%.

At first, these percentages can seem very low (hence good), but it is extremely important to take into account the full traffic load targeting a given web server. For instance, concerning the Google web server, the false positive rate after the analysis of a day of HTTP requests (490.704 queries) was 0.04%. As a consequence there were nearly five thousands (4944) alarms that in fact were not anomalies. For some organizations, five thousands false alerts per day can be impossible to handle.

In the following sections, the models implemented in [1] are described in details. Part of them has been implemented also in the system I developed, with some major differences, needed to overcome the limitations encountered analyzing real world HTTP traffic.

Attribute Length Model

The length of a query attribute can be used to detect signs of infections, especially if a certain attribute accepts only fixed-length values. The goal of this model is to approximate the actual but unknown distribution of the parameter lengths and detect instances that significantly deviate from the observed normal behavior [1].

To achieve this objective, the model computes the mean μ and the variance σ^2 of the attribute lengths of all the values observed during the training phase for a given attribute. The creation of the model has a very low cost, proportional to the number of queries processed in the training phase.

The model computes the anomaly score for a certain value making use of the *Chebyshev's inequality*. This is a measure of distance that guarantees that there exists very few elements of a distribution for which the difference between their value and the mean exceeds a certain threshold. More in details, the probability p that the difference between a random variable x and the mean μ exceeds the threshold t is less than the ratio between the variance σ^2 and the square of the threshold, as illustrated by the formula:

$$p(|x - \mu| > t) < \frac{\sigma^2}{t^2} \tag{3.1}$$

When the length l of a value is very far from the mean, the probability p of a legitimate sample having a length greater than l should be very small. Therefore, the authors of the paper defined the threshold t as the difference between the length l of the attribute value and the mean μ. Thus, the previous formula becomes:

$$p(|x - \mu| > |l - \mu|) < \frac{\sigma^2}{(l - \mu)^2} \tag{3.2}$$

The Chebyshev's inequality presents several advantages:

- First of all it can be applied to any probability distribution in which the mean and variance are defined, so it is suitable for almost any kind of data.
- It is different from most of the techniques that try to identify a range of acceptable lengths, because this method takes into account the variance of the distribution observed during the training phase and it returns a probability value (not only a boolean anomalous/benign).
- It is quite efficient because it requires only to compute the length of the input and to perform a simple computation.
- It computes a very loose upper bound meaning that it has an high degree of tolerance. In many situations this can be inadmissible but, since the distributions of the attribute values usually have a large variance, in this case it can be useful to report only the significant anomalies.

. This model can be particularly efficient in identifying those attacks that require a big payload, such as Buffer Overflows or Cross-Site Scripting attacks.

Attribute Character Distribution Model

This model profiles the character distribution of a regular attribute. Often, attribute values have a constant structure and the used characters are taken from a small subset of the 256 possible values. For instance, it is quite common that attribute values contain mostly letters and few special characters. Therefore, legitimate inputs usually have a character distribution that follows a specific pattern.

To identify this pattern, the model computes the relative character frequency sorted in descending order for each attribute value seen during the training phase. Thus, for each value the model stores an array with 256 entries, one for each possible byte (that in most situations corresponds to a character). By sorting in descending order, the relationship between the individual character and its relative frequency is lost (i.e. it is impossible to know the number of occurrences of a certain character). The authors observed that, for legitimate values, the obtained distributions decrease slowly, without any character having a predominant frequency.

At the end of the training phase, the model derives the *Idealized Character Distribution* (ICD) for a particular attribute by computing the average of all the collected characters distributions. Then it starts computing the anomaly scores for validation and testing phase.

This model makes use of a statistical test to assess if a given value is malicious or benign (i.e. if it belongs to the observed ICD or not), the variant of the Pearson's χ^2-test for the *goodness of fit*. The Pearson's χ^2-test is a widely used statistical procedure and, in particular, the variant for the goodness of fit establishes whether an observed frequency distribution differs from a theoretical distribution. This statistical measure tests the validity of the *null hypothesis*, stating if the frequency distribution of certain events observed in a sample is consistent with a particular theoretical distribution.. It applies to categorical data only and it requires that every level of the categorical variable in analysis has a minimum number of occurrences (in literature the recommended value is at least 5).

The test involves the following steps:

1. Find the degrees of freedom *df*. This parameter identifies the number of independent variables and it is equal to the number of levels (k) of the categorical variable minus 1:

$$df = k - 1 \tag{3.3}$$

2. Compute the expected frequencies (E). For each level of the categorical variable, the expected frequency is computed as the sample size times the related proportion from the theoretical distribution.

$$E_i = l * n_i \tag{3.4}$$

   Where E_i is the expected frequency of the categorical data at level i, l is the length of the sample value and n_i is the proportion from the theoretical distribution.

3. Compute the χ^2 value. This is the normalized sum of squared deviations between the observed frequencies O and the expected frequencies E:

$$\chi^2 = \sum_{i=0}^{i<k} \frac{(O_i - E_i)}{E_i} \tag{3.5}$$

   Where O_i is the observed frequency of the categorical variable at level i.

4. Derive the probability value p, representing the probability to observe a distribution as extreme as the one with the computed χ^2 value. This value is taken from the χ^2 distribution using χ^2 and *df* and looking up in a pre-defined table. The higher p the higher the confidence that the value under analysis is taken from the expected distribution.

Since the χ^2-test can be applied to categorical data only, the authors divide the function domain of a character distribution into six intervals. The choice of the bins is arbitrary and the values were aggregated as follows:

1. [0]
2. [1, 3]
3. [4, 6]

4. [7, 11]
5. [12, 15]
6. [16, 255]

Even though this choice is completely subjective, it takes into account the fact that lower indices in the ICD contains an higher number of occurrences, because the character distributions are sorted in descending order.

When the input value is provided to the model, it first computes its absolute character frequency (sorted in descending order). Then, the observed frequencies O are obtained by aggregating the elements according to the scheme defined above. The expected frequencies E, instead, are computed by multiplying each bin of the ICD times the length of the attribute value under analysis. After this step, the χ^2-value can be measured. In this scenario, the degrees of freedom df are five (the number of bins minus one). At this point, the model has all the information to retrieve the probability value p, that represents the anomaly score of the attribute value.

Like the Attribute Length Model, this model is not expensive to build and apply, as it only requires to compute the character frequency of the input value and the χ^2-test. The cost is linear in the length of the input value for the the first operation and constant for the second.

This model is mainly intended for attacks that send binary data (such as Buffer Overflows) or that repeat the same character many times (such as in Directory Traversal exploits). Since most of the attribute values are made of printable characters, it is likely that the character distributions in presence of these kind of attacks differs broadly from the ones of regular attribute values.

However, one major drawback of this technique is that the χ^2-test is very unreliable when the bins have low values. In real world scenarios, it is not uncommon that some bins have zero or very low values, especially for attributes that have short length, and this was one of the challenges I faced when developing my system.

Structural Inference Model

This is a very specific model of a query attribute that profiles values in order to build the regular grammar of legitimate inputs. The grammar should be able to generate at least all the values present in the training set, but it can also be generalised to produce arbitrary strings.

The approached followed by the authors is to generalize the grammar as long as it seems to be "reasonable" and stop before too much structural information is lost. The notion of "reasonable generalization" is specified with the help of Markov models and Bayesian probability [1].

The cost of building this model without any optimization is very high, $O(n * l)^3$, with n the number of values analyzed during the training phase and l their maximum value. Such a complexity makes this model impossible to use as is. Therefore, the authors applied a number of different optimizations (such as the *Viterbi path approximation*) reducing the complexity to $O(n * l^2)$. This is much more affordable for moderate values of l, even though still expensive.

The Structural Inference is a very effective model that is particularly thought to target those scenarios in which the Attribute Length and the Attribute Character Distribution models cannot detect the maliciousness. In fact, a powerful attacker can be able to hide his attack inside attribute values that looks perfectly normal concerning the length and the byte distribution. However, by knowing the structure of the attribute legitimate values, even a well-crafted attack can be spotted.

The major drawback of this technique is the high building cost, even with all the optimizations, that can become prohibitive if l is too large. This is the reason why this model is hard to use in scenarios where the traffic load can be huge and it is critical to be very responsive.

Token Finder Model

The goal of this model is to assess if a query attribute accepts only a finite number of values (i.e. the attribute belongs to an enumeration). It is not uncommon that certain attributes can assume only few possible values (for instance a boolean flag can only be true or false).

Therefore, the Token Finder is used to detect attacks that send an unexpected attribute value. Of course, if for a certain attribute an enumeration cannot be established, no alert is raised by this model.

The classification of an attribute as enumeration or random is based on the assumption that the number of possible values is bound by a threshold t. During the training phase, if the number of a feature values grows proportionally with the number of occurrences of the feature, then it is likely that the attribute is not an enumeration. More formally, the model computes the Pearson Correlation Coefficient ρ between the functions f and g of the occurrences of a certain attribute a.

The Pearson Correlation Coefficient is a measure of the linear correlation between two variables (in this case the functions f and g). It can assume values between -1 and +1, where:

- A correlation of -1 indicates that the two variables are perfectly negatively correlated. If one increases in value, the other decreases proportionally.
- A correlation of 0 indicates that the two variable are not correlated together.
- A correlation of 1 indicates that the two variables are perfectly correlated. They both increase and decrease with the same proportion.

The Pearson Correlation Coefficient and the functions f and g are defined by the formulas:

$$f(x) = x \tag{3.6}$$

$$g(x) = \begin{cases} g(x-1)+1 & \text{if the xth value for } a \text{ is new} \\ g(x-1)+1 & \text{if the xth value for } a \text{ is new} \\ 0 & \text{if x = 0} \end{cases} \tag{3.7}$$

$$\rho = \frac{Cov(f,g)}{\sigma_f * \sigma_g} \tag{3.8}$$

Where $Cov(f,g)$ is the covariance between f and g, and σ_f, σ_g are the corresponding standard deviations.

The functions f and g are computed during the training phase. The first one reflects the increasing number of analyzed parameters (it is just a counter of the occurrences). The second one, instead, increases if a value has not been seen before and decreases otherwise. Thus, if the same value recurs many times, g decreases.

At the end of the training phase, ρ is computed. If it less than 0, an enumeration is assumed because the observed trend is that as more attribute values are processed, few new values are registered. Otherwise, if ρ is grater than 0, this means that there is an high variability in the values assumed by the attribute under analysis.

If the attribute is considered as an enumeration, the set of all the observed values is stored to be used during the detection phase. The detection simply consists in checking if the input value belongs to the enumeration (if the attribute is not classified as random). If it does not, an alert is reported.

The cost of building this model is low, because it only requires to compute the Pearson Correlation Coefficient and it depends on the number of queries in the training set. The detection is a simple table lookup.

Attribute Presence/Absence Model

While the previous models were specific for a single query attribute, this one (and the following) profiles the behavior of the whole query, taking into account multiple attributes at once.

Usually, web server are not accessed directly by the user, i.e. an user does not write the URL in the browser himself, but it is a client-side program or script that does the job on his behalf. For example, when an user submits a form, the information are automatically sent to the web server in a predefined way. This results in a high regularity of the traffic, meaning that for a certain web resource the number of parameters and their order is practically standard. On the contrary, an attacker usually crafts by hand the URL to send to the web server and does not pay attention to the order or the completeness of the parameters.

Therefore, the purpose of this model is to learn which are the legitimate parameter for a certain web resource. If a required attribute is missing or if two mutually exclusive parameters are present it can be a sign of attack.

During the training phase, the model simply records each group of parameters present in a certain web request. Then, in detection phase, it checks if the set of parameters for the input query has been seen in the training phase. If it does not, an alert is reported. Notice that there is not a validation phase in this case.

Building this model is very efficient because it only requires to store a set of value for each processed input. In the same way, the detection cost it is low, as it requires only one table lookup.

The Attribute Presence/Absence targets those situations where an attacker probes a web-server by sending incomplete or malformed requests. However, there might be some false positive, because it is not always feasible to learn during the training phase all the possible sets of parameters suitable for a query. Moreover, there are web-servers built in such a way that a query can contain one or more parameters randomly generated. In these cases, it is impossible to check if a parameter is required or not because it constantly changes and this would always lead to false alarms.

Attribute Order Model

This model is strictly related to the previous one. Its goal is to profile the parameters order for a certain query, under the assumption that server side programs are invoked following a standard format.

An attribute a_i of a program (i.e. a web server resource path) precedes another attribute a_j when a_i and a_j appear together in the parameter list of at least one query and a_i comes before a_j in the ordered list of attributes of all queries where they appear together [1]. Therefore, the order constraints is defined as a set of attribute pairs O:

$$O = (a_i, a_j) : a_i \text{ precedes } a_j \text{ and } a_i, a_j \in (S_{q_j} : \forall j = 1, ..., n) \tag{3.9}$$

where S_{q_j} is the set of attribute of query q_j and n is the corresponding number of queries.

During the training phase, the set O is built in two steps:

1. First the graph G is created for a certain program. For each distinct attribute a_i, G has a corresponding vertex v_i. For every processed query, the model collects the ordered list of attributes (a_1, ... ,a_i). Then, for each attribute pair in the list (a_i, a_j, with $i \neq j$), a directed edge $v_i \rightarrow v_j$ is inserted in the graph.

2. At the end of the first step, G contains all the order constraints defined by the queries in the training set. A constraint between two attributes (a_i, a_j) is represented by either a direct edge between the corresponding vertices (v_i, v_j), either by a path that links together the two vertices.
   However the graph built in this way can contain cycles due to different order constraints derived from different queries, therefore another step is required to compute O. In fact, the presence of a cycle would make impossible to check whether a test query satisfies the order

dependencies or not because it would cause an infinite loop between the vertices of the graph. The solution to this problem is obtained applying to the graph G the *Tarjan*'s algorithm, a linear-time complexity technique used to find the Strongly Directed Components (SCC) of a direct graph. A strongly connected component of a directed graph is a maximal strongly connected sub-graph and a sub-graph is said to be strongly connected if every vertex is reachable from every other vertex. Figure 3.3 shows an example of the strongly directed components (represented by the two dotted squares) of a direct graph.
Once the SCCs are obtained, an acyclic graph is derived by removing in each component all edges connecting vertices of the same SCC. Finally, the set O is built starting from the acyclic graph by simply enumerating for each vertex v_i all its reachable nodes (v_g, ... ,v_h) and adding the corresponding attribute pairs ((a_i, a_j), ... ,(a_i, a_h)) to the set.

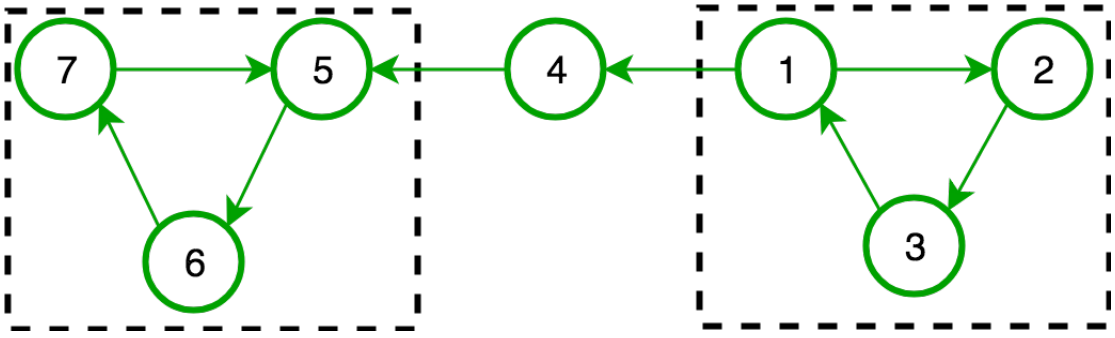

Figure 3.3: The SCCs of a graph.

During the detection phase, the model analyzes all the attribute pairs ((a_i, a_j), with $i \neq j$) of the input query in order to find a violation of the order constraints defined in O. A violation occurs if, for any attribute pair (a_i, a_j), the corresponding pair with switched elements (a_j, a_i) is present in O. In this case the model raises an alert.

Similarly to the Attribute Presence/Absence, the Attribute Order is meant to detect those attacks in which the attacker crafts by hand the malicious request and he can introduce a discrepancy in the usual attribute order for a given query.

Access Frequency Model

Unlike the classifiers presented so far, this model (and the following two) does not profile a single query or a single attribute, but takes into account multiple queries at once.

The objective here is to monitor the access frequency of a certain web server program. Different resource paths are queried with different frequencies, but the general access pattern should remain practically constant over time.

For each program, two types of access frequency are profiled: the absolute frequency of all accesses and the access frequency for each client (identified by the IP address). These two separate metrics have been chosen to build a complete model of the access pattern.

On one hand it is in fact possible to have some applications that are invoked very often in general but not so much from a single client. An example of this scenario can be a login page, which is visited once per each client and very often overall because it is the first page used to access a web server.

On the other hand, there are also programs that are invoked very often by a single client but they are not very popular globally. This could be the case of a search page, that is not visited regularly but it is rather accessed in burst by a client looking for a certain information. A change in these access patterns can be a symptom of attack.

During the training phase, the time interval identified by timestamp of the first query and the timestamp of the last one in the set is split in smaller fixed-size periods (for instance 10 seconds per

period). Then, the absolute access frequency and the number of accesses per client are measured in each of these periods. At the end of this step the model obtains two distributions (one for each access pattern) and computes their mean μ and variance σ.

During the testing phase the time is divided in intervals of the same size used for the training phase. For each query the model computes the absolute access frequency and the access frequency per client, deriving the two testing distribution. These patterns are compared with the corresponding training distributions by means of the *Chebyshev's Inequality*, the same technique applied for the Attribute Length Model in section 3.2.1. The final anomaly score is derived by computing the average of the two probability values returned by the two *Chebyshev's Inequality* measures.

The cost of both of learning and detection is proportional to the number of requests processed in the training and in the detection phase, respectively.

This model is intended stop an attacker probing a web application for vulnerabilities or attacks such as bruteforce exploits, where the attacker sends many web requests trying to guess the value of a certain parameter.

Inter-Request Time Delay Model

This is an example of another kind of model that profiles the client access patterns towards a web application. However, unlike the previous classifier, this one monitors the delays between the requests sent by each client.

During the training phase, the regular distribution of the time intervals between consecutive requests is created. To do so, for each client the model stores the time delays between successive queries. Then, similarly to what is done in the Attribute Character Distribution Model in section 3.2.1, the delays are aggregated in small bins. Finally, the regular distribution is obtained by averaging the values of the bins of all the observed clients.

During the detection phase, a distribution of time delays is created for each client. Next, each distribution is compared with the training distribution by means of the Pearson's χ^2-test, used also in the Attribute Character Distribution Model. The anomaly score that is returned by this model depends on two factors: the likelihood that an observed distribution is a sample from the learned expected distribution as described above and, additionally, the number of requests which have been monitored from a specific client for that application [1].

The first aspect is related to how anomalous the testing distribution is with respect to the training one, while the other one defines with how much confidence we should consider this anomaly score. The latter is very important because, if a client sends only few requests, the corresponding distribution could be considered anomalous but there is not enough data to be sure of it.

Therefore, the anomaly score is scaled by a factor that takes into account the number of processed requests for a client. This factor increases as the number of queries increments and it eventually will reach a value of 1 and it will be discarded (when the sample data will be big enough to have high confidence in the predictions).

This model can be built and applied efficiently, as its cost is linear in the number of processed queries. It raises an alert when a deviation from the computed regular distribution is detected. This can happen especially in case of some probing attacks, where the web requests are sent at regular intervals, because this pattern is not common to most of legitimate clients (which instead show a more variable behavior).

Invocation Order Model

This is the last model presented in [1] and it analyzes the traffic at the highest level of granularity.

In general, the programs exposed by a web-server are accessed by a client following a well-defined pattern. Consider, for instance, an e-commerce web application such as Amazon. If a customer wants to buy an item, he first needs to login with his credentials before he can complete

the purchase in the appropriate area. Therefore, one could expect that the client first visits the login page, then it looks for an item and finally it reaches the purchasing page.

The purpose of this model is to learn the natural order of invocations of different web-application of a certain web server. In this way, it is possible to derive the structure of a regular session for a client. Deviations from this pattern cause the raise of an alert.

During the training phase, the model builds a session by grouping together the program invocations done in a certain time interval by a client (always identified by the IP address). This is different from the aggregation technique of the other models that was done always on a per resource path basis (instead of per client).

Given all the training sessions, the model infers the structure of a regular session using the same method as the Structural Inference Model in section 3.2.1. The only difference is that in this case the inputs are series of program accesses instead of sequences of attribute characters.

During the testing phase, a certain query is associated to its session S. If S belongs to the learned pattern the query is considered benign, otherwise it is deemed malicious.

Since this model is analogous to the Structural Inference, it shares with it the same (quite expensive) costs for creation and application.

The Invocation Order mainly aims at detecting attacks to the application logic. It should be able to recognize situations in which an attacker tries to bypass a credential check (such as a login page) to access privileged pages directly.

3.3 TokDoc: A Self-Healing Web Application Firewall

TokDoc is a system that integrates many of the concepts presented in [1] and it also proposes some very interesting extensions. It is defined by the authors as a reverse HTTP proxy.

I introduced the concept of forward proxy server, or simply proxy, in section 2.3.2. The main difference between a forward and a reverse proxy is that while the former is a server situated in front of the client that delivers a client's request to the target web server, the latter works in the opposite way, sitting in front of a web server and sending to it the requests coming from the clients. The sample traffic flow of these two kinds of proxy is illustrated in figures 3.4 and 3.5 respectively.

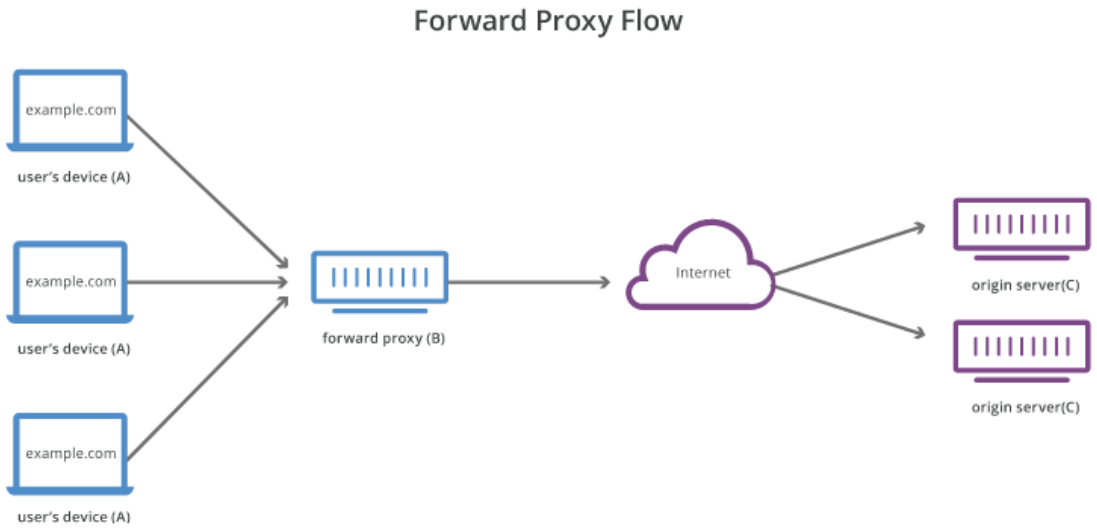

Figure 3.4: Traffic flow of a forward proxy.

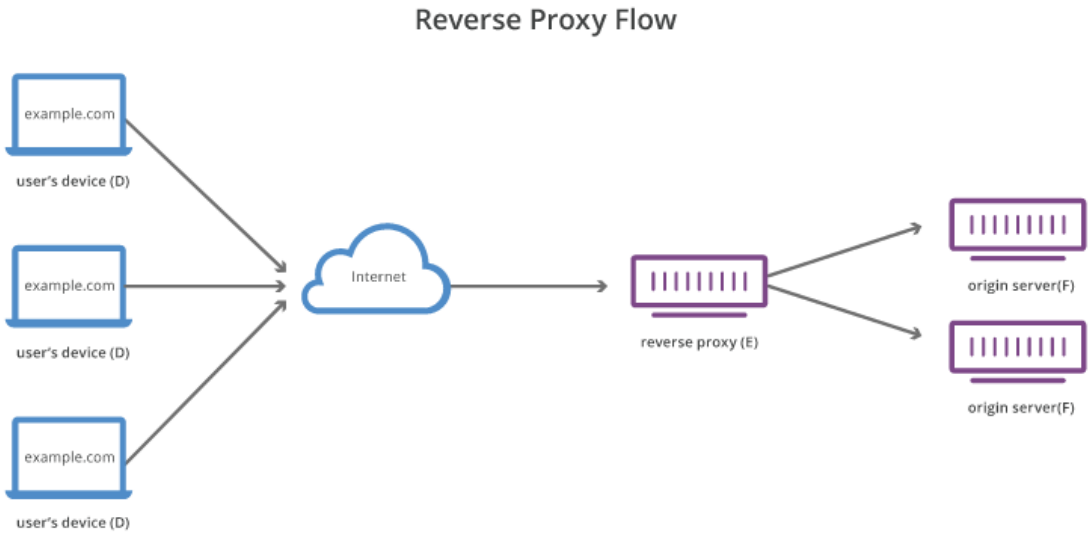

Figure 3.5: Traffic flow of a reverse proxy.

A reverse proxy has three main applications:

- **Security**: it allows to perform traffic filtering (therefore it can block potential malicious requests coming from the clients) and it hides the web server's IP address, making more difficult to perform Denial of Service Attacks (DOS).
- **Load Balancing**: a popular web application can be distributed among multiple servers. The clients requests will arrive to the proxy that will divide the traffic evenly to the servers, avoiding overloading.
- **Caching**: it can temporarily store some documents. In this way, if a client requests always the same content, this can be provided directly by the proxy, thus improving the performance and reducing the load on the web server.

As in [1], TokDoc performs anomaly detection on HTTP requests based on feature models. However, it extends the analysis with respect to the work of Krueger et al. because, while in [1] an alert raised by a model affected the whole web request (regardless of the granularity level at which the model was operating), here each decision interests only the *token* subject of the analysis.

More in details, every request is parsed into token-value pairs according to a given heuristic and then the anomaly detectors are applied to detect signs of attacks for that specific token. If the processed value is deemed malicious, TokDoc employs a mangling technique to make the anomaly harmless and it delivers the HTTP request to the recipient.

The mangling techniques are named *Healing Actions*. Beside detecting and reporting an intrusion, they can also neutralize an attack instead of simply dropping the malicious web request. This not only improves detection accuracy but makes decisions more fault-tolerant, since the replacement of content with a suitable alternative in certain cases does not harm even if it has been wrongly classified as malicious [2]. An Healing Action is automatically assigned to a certain token according to token-specific rules during the so called *Setup* procedure.

Finally, one major advantage in comparison with [1] is that the system is built in such a way that it does not require a clean training set to make the models learning the regular traffic targeting a web server, because the implemented technique should be robust against contaminated data.

TokDoc is therefore made of three components:

- **Token Types**: the authors define four token types, based on their analysis of real HTTP traffic.

- **Anomaly Detectors**: anomaly detection techniques associated to certain token types during a setup phase.the Setup procedure.
- **Healing Actions**: the authors propose for different techniques, depending on the token type and on the desired level of protection.

These elements are discussed more in details in the following sections.

3.3.1 Token Types

TokDoc parses every received HTTP request in token-value pairs. It considers as tokens all the GET parameters (like in [1]), but it also extends the analysis to the URI path, the POST parameters and all the HTTP header fields.

Naturally, the distribution of token values is very diverse, thus they are classified in four categories, according to their properties:

- **Constants**: the simplest case of token, in which the value is always the same. An example is the header field *user-agent* when monitoring a certain session.
- **Enumerations**: these are tokens that can accept only a small set of values, such as the header field *accept-language*.
- **Machine input**: this third type of tokens comprises machine-generated data, such as session numbers, identifiers and cookies [2].
- **Human Input**: this category includes all the human-provided inputs, such as form values. This is the most difficult type to monitor because the data can be anything, without restrictions.

The peculiar characteristics of the token types are used to properly assign the anomaly detectors and healing actions, as described next.

3.3.2 Anomaly Detectors

TokDoc applies anomaly detection algorithms to every token of an HTTP request. The anomaly detectors are built following the ideas presented by Kruegel et al. in [1]. The main difference is that, unlike in [1], the decisions taken by a certain classifier do not affect the whole HTTP request but only the corresponding token.

There are four anomaly detectors:

- LIST
- N-gram Centroid Anomaly Detector (NCAD).
- Markov Chain Anomaly Detector (MCAD).
- Length Anomaly Detector (LAD).

The LIST detector is the default one for constants and enumerations, while the others are automatically assigned to the corresponding token type during the Setup procedure (described later).

LIST

This is the simplest classifier and it is very similar to the Token Finder Model presented in [1].

It just records all the different values observed during the training phase for a specific token. Then, when working in detection mode, if the value under analysis has not been seen in the training data, it is deemed suspicious.

N-gram Centroid Anomaly Detector (NCAD)

N-gram models are very popular in security applications. An n-gram is a sequence of n *items* taken from a given sample input. This technique is commonly used in text or speech processing, therefore an n-gram is often defined as a sequence of n-words. For instance, a bigram is a two-word sequence, a trigram is a three-word sequence and so on.

An N-gram model is a probabilistic metric that predicts the occurrence of an item based on the occurrence of its N - 1 previous items. It answer the question: how far back in the history of a sequence of items should we go to predict the next item? For instance, picking up the previous example, a bigram model (N = 2) predicts the occurrence of a word given only its previous word (as N - 1 = 1). Similarly, a trigram model (N = 3) predicts the occurrence of a word based on its previous two words (as N - 1 = 2 in this case).

TokDoc performs an n-gram analysis in which the items are not words but bytes sequences. Given the set of all possible n-grams S = {0,...,255}^{n}, the authors define the embedding function ϕ for a token value x as :

$$\phi(x) = (\phi_s(x))_{s \in S} \in \Re^{|S|} \tag{3.10}$$

Where $\phi_s(x)$ returns 1 if the n-gram s is contained in x and 0 otherwise.

The parameter n is critical for the model as it affects the trade-off between performance and precision. The higher the more precise is the analysis because the model has more context to make the prediction. On the other hand, the vector space induced by the embedding of n-grams grows exponentially with n [2].

Given two embedding vectors $\phi(x)$ and $\phi(z)$, they are first normalized to one to eliminate the length dependency and then they are compared measuring the Euclidean distance *d(x,z)*:

$$d(x, z) = ||\phi(x) - \phi(z)||_2 = \sqrt{\sum_{s \in S} |\phi_s(x) - \phi_s(z)|^2} \tag{3.11}$$

The NCAD model defines the embedding vector μ of a regular token as the arithmetic mean of the embedding vectors computed from the training values of that token:

$$\mu = \frac{1}{k} \sum_{i=1}^{k} \phi(x_i) \tag{3.12}$$

where k is the number of training elements and x_i is value i for token x.

Once the model has μ it can derive the anomaly score as:

$$score_{NCAD} = \begin{cases} \text{normal} & \text{if d}(\mu, \text{x}) \leq t_a \\ \text{anomaly} & \text{otherwise} \end{cases} \tag{3.13}$$

where t_a is a threshold defined during the Setup procedure.

Markov Chain Anomaly Detector (MCAD)

Markov chains have been widely used in literature for security purposes. A Markov chain is a mathematical system that experiences transitions from one state to another according to certain probabilistic rules. It is an example of *stochastic process*, with the addition that it satisfies the Markov property: no matter how the process arrived at its present state, the possible future states are fixed. This means that the probability of going from one state to another one does not depend on the previous visited states but only on the current state and the time elapsed.

Markov chains have many applications as statistical models. They are usually modeled as finite state machines and the state space, or set of all possible states, can be anything (letters, numbers or even market stocks). Figure 3.6 shows an example of Markov chain in the form of a state

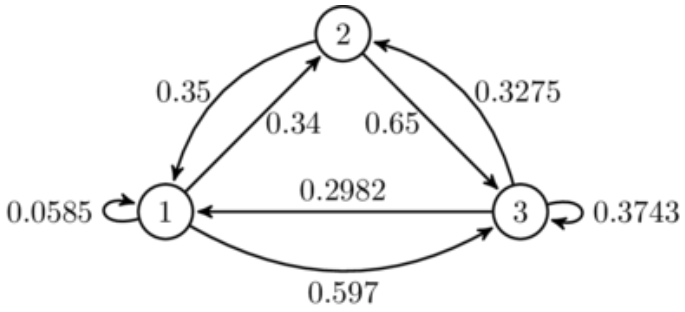

Figure 3.6: An example of Markov Chain with three states.

machine, where each node is a state and each edge represents a transition with the corresponding probability.

TokDoc defines the state space as the 256 possible byte values. Each state can have 256 possible transitions (we need to consider loop transitions, back to the same state). The state transitions probabilities are learnt by monitoring the transition frequencies from consecutive byte values in the training data. In this way, for each token, TokDoc builds a transition table with $256^2 + 256$ rows and columns. Notice that the 256 additional entries represent the dummy start state of each byte.

Given the transition table, the probability of a token value x of length n for a Markov Chain C is:

$$P(x|C) = P(X_1 = x[1]) \prod_{i=1}^{n} P(X_{i+1} = x[i+1]|X_i = x[i]) \tag{3.14}$$

where *x[i]* represents the *i-th* byte in the token value x [2].

The authors did not use any length normalization technique in order to take in to account both the length and the content of the analyzed value. Once computed $P(x|C)$, the anomaly score is obtained as:

$$score_{MCAD} = \begin{cases} \text{normal} & \text{if } P(x|C) \geq p_a \\ \text{anomaly} & \text{otherwise} \end{cases} \tag{3.15}$$

where p_a is an anomaly threshold computed during the Setup procedure.

Length Anomaly Detector (LAD)

This detector achieves the same goal as the Attribute Length Model in [1] (identify malicious values based on their length) but adopts a different metric. In fact, the Chebyshev's Inequality used in [1] can be very unreliable, especially when the training set is small. Therefore, TokDoc tries to find a solution efficient also when data availability is low.

Given a predefined significance level α_{LAD}, the authors estimate the $1 - \alpha_{LAD}$ quantile of the length distribution of the train and validation data, namely $L_{1-\alpha_{LAD}}$. Then they construct a confidence interval for $L_{1-\alpha_{LAD}}$ by first calculating σ, the bootstrap estimate of the standard error of $L_{1-\alpha_{LAD}}$ [2], obtaining the interval:

$$I = (L_{1-\alpha_{LAD}} - c\sigma, L_{1-\alpha_{LAD}} + c\sigma) \tag{3.16}$$

where c is a constant chosen in such a way that I has a probability coverage of $1 - \alpha_{LAD}$.

The upper bound of the interval I is used as threshold, therefore the anomaly score for a token x is computed as:

$$score_{LAD} = \begin{cases} \text{normal} & \text{if len(x)} \leq L_{1-\alpha_{LAD}} + c\sigma \\ \text{anomaly} & \text{otherwise} \end{cases} \tag{3.17}$$

This detector is less complex than the NCAD and MCAD detectors and it is used only for tokens that, during the learning period, are not able to collect enough data to train the other two anomaly detectors.

3.3.3 Healing Actions

Once a token value is deemed to be malicious, TokDoc can apply an healing action to "sanitize" the input and send an harmless web request to the server.

This approach is much better than simply dropping the whole anomalous request, because often the "healed" token value does not affect the response from the web server. Therefore, in case TokDoc wrongly flags a value as suspicious (i.e. it generates a false positive), there still is the chance that the request reaches the server and the response is correct.

The choice of the particular healing action is tightly correlated with the token type and the assignment is done during the Setup procedure. TokDoc implements four healing actions:

- **Drop**: this is the most conservative measure, that entails the suspicious token to be discarded. Notice that, even if extreme, this action is still better than dropping the whole request. It is the default option for each token assigned to a LAD detector.
- **Encode**: this approach encodes the suspicious value with HTML entities. In this way, most of the web attacks based on Cross-Site scripting and SQL injection are neutralized because the dangerous characters are escaped. Moreover, this technique does not affect the majority of the web server because they are usually able to handle the additional encoding layer.
- **Freq**: this action replaces the anomalous value with the most frequent one for the particular token. This is the default choice for tokens using the LIST detector, such as constant and enumeration tokens.
- **Near**: the most complex action, that replace the malicious token value with its nearest-neighbor from the training set. It is associated with tokens assigned to NCAD and MCAD detectors.

3.3.4 The Setup procedure

The setup of TokDoc strongly depends on the initial available data set. The more web requests are available for training and validation the better.

A detector is automatically assigned to a particular token following a well-defined workflow:

1. The HTTP requests in the input data set are grouped according to the target web application exposed by the web server (as in [1]), allowing for the creation of service-specific models.
2. The pool related to a particular service is further divided in two smaller groups: the training set to learn the models and the validation set to set up the thresholds (again based on [1]).
3. After the learning phase, each token of the original data set is assigned to its detector by making use of both a structural and a statistical test, according to the following constraints:
   - If the training set contains less than 50 samples for the current token, this is assigned to the LAD detector. This choice is supported by the fact that the other detectors require much more samples in order to be effective.
   - If more than 50 samples are available, TokDoc assesses if the token is an enumeration or not. First it checks if less than 10 unique values have been observed. If this structural test is passed, the χ^2-test is used as statistical tool to compute the probability that new unseen values will occur in the future. If this probability is below a predefined threshold, the token is linked with the LIST detector.
   - Otherwise, the remaining choices are between the NCAD and the MCAD detectors. At this step, the structural test simply measures the median length of the token values. If it is higher or equal than 5, TokDoc computes the probability value P:

$$P(||X - \mu|| \geq d_{max}) = 0 \tag{3.18}$$

where X is the training data for the token under test, μ is the NCAD centroid and d_{max} is the maximum distance from μ, defined as:

$$d_{max} = \sqrt{||\mu||^2 + 1} \tag{3.19}$$

If $P(||X - \mu|| \geq d_{max})$ is equal to 0, NCAD is chosen, otherwise MCAD is assigned to the current token.

4. After the assignation is completed, the anomaly thresholds for both NCAD and MCAD are computed. The threshold is set to be the maximal distance observed in the validation set for the NCAD detector, while the minimal probability is used for MCAD.
   The threshold is defined after a semi-automatic outlier adjustment: all values of the validation data set are ordered by the according output of the detector (descending distances to the mean for NCAD and ascending probabilities for MCAD) and a system administrator decides whether the extremal value is a real, user-generated sample or a malicious token value [2].

5. Before deployment, the system administrator can manually modify the healed action associated to the token. For instance, privacy sensitive data such as passwords and cookies should not use the *Near* healing action but should instead drop the token completely.

6. After deployment, if the detector of a certain token produces too many false positives, the system administrator can decide to trigger again the setup procedure for that token.

TokDoc implements a Setup procedure that is completely data-driven. However, it requires the presence of a system administrator to properly prepare the input data set. As I will detail better in the next chapter, the anomaly detector I realized is completely autonomous: it is able to automatically build its models and it does not require any pre-processing step (neither manually as in [1], neither semi-automatic as in [2])

Chapter 4

Methodology

HTTPServer is an anomaly detector system designed to identify and report unusual parameters in the HTTP queries targeting a web server to protect. It is integrated into LLAnta, Lastline's network analysis system and it works in cooperation with many other anomaly detectors that target different threats, such as port scans, TLS and DNS anomalies.

The basic structure reproduces the ideas presented in [1] and [2]: it analyzes incoming HTTP GET requests, create machine learning models to profile the behavior of the specific web server and it raises alerts if something looks suspicious. The produced alerts will be sent to the LLAnta Manager for further processing and, eventually, they will be shown in the customer user interface.

HTTPServer proposes different, more efficient solutions to some of the concepts introduced in the literature and it adds several interesting features on top of them, with the goal to contribute in creating a more precise and reliable product.

HTTPServer contains four components interacting with each other:

- **Anomaly Classifier**: the basic unit of the whole anomaly detector system, in charge of building a model of the server behavior according to a specific heuristic. HTTPServer implements many different classifiers, each of them devised to address a particular threat.
- **Plugin**: a LLAnta plugin, as described in section 2.4. It processes the web requests targeting the web server and and manages the classifiers.
- **Detector**: a LLAnta detector, that decides which web request was actually malicious and reports the alerts to the LLAnta Manager.
- **Context**: a LLAnta context, that allows to store information on disk and permits the communication between the plugin and the detector.

In this chapter I am going to describe in details the overall architecture, how the system works and the reasons behind the design choices I made. In the next sections I outline the mode of operation of each component following a bottom-up approach. I will therefore start introducing the fundamental working units (the classifiers) and how they combine together to build a more complex system.

The last section of the chapter, instead, will focus on the testing mechanisms I adopted to make sure everything worked as expected before deploying the system in the real world. Writing tests is a core component in developing new products. It is important to build a suite that is able to emulate as close as possible existent working environments because in this way it is possible to reduce to the minimum the number of bugs the system has when deployed. It also helps the development of the application itself and adding new features. For instance, if by introducing a new element to the system the tests of an unrelated component fail, there is clearly a logic bug somewhere in the code.

4.1 Anomaly Classifiers

An anomaly classifier is the equivalent of a model in [1] and of an anomaly detector in [2]. It profiles the inbound traffic, employing and renovating the three-step approach introduced in [1]:

1. **Training**.
2. **Validation**.
3. **Testing**.

The main difference, however, is that each anomaly classifier automatically goes from one operation mode to the other one, without the need for any human interaction or offline processing. In fact, both training and validation are done monitoring directly the customer traffic. This is possible because the implemented system is resistant against contaminated data, that are expected when analyzing real world traffic. As a consequence, there is no requirement for any clean data set to build the models (while is needed in [1]), nor any time consuming pre-processing step is necessary (like in [2]).

HTTPServer implements three anomaly classifiers:

- **LengthClassifier**: the purpose is to identify anomalous parameter values based on their length. This kind of detector has already been proposed both in [1] and [2], but my implementation uses completely different techniques.
- **DistributionClassifier**: the goal of this model is to detect malicious values based on their byte distribution. It is quite similar to the Attribute Character Distribution Model presented in [1] but it introduces some major improvements.
- **DataTypeClassifier**: this kind of anomaly classifier is new with respect to both [1] and [2]. It tries to determine the type of the parameter under analysis and it flags as malicious any value that does not belong to the learned data class.

On top of these three basic anomaly classifiers, HTTPServer introduce also the AttributeModel class, that aggregates and combine the above classifiers (from now on *sub-classifiers*, because they are working units of an AttributeModel) to provide a reliable, tamper-resistant model of an HTTP parameter. It determines the regular behavior of a certain attribute based on the heuristics provided by its sub-classifiers and it is the component that actually predicts if a value is benign or not.

The AttributeModel objects are managed by HTTPServerPlugin, that processes all the incoming web requests and sets up HTTPServerContext for HTTPServerDetector (as described next in the corresponding sections).

All the anomaly classifiers (i.e. AttributeModel and sub-classifiers) implement a well-defined interface represented by the *AbstractClassifier* abstract class. This design choice allows to standardize the way each classifier should operate, regardless from the particular metrics in use. Having an interface to be compliant to greatly increases the modularity of the code.

Each member of the interface exposes the same API (Application Program Interface) to the outside, meaning that the interaction with any module is done always invoking the same methods, even if the inner implementation is completely different from one component to another. Of course a certain unit can define other methods needed to accomplish its working target. The only restriction is that all the methods defined in the interface must be implemented.

This design pattern makes easier to extend the system (it is only needed that the new element exposes the same API) and widely improves the code readability because the structure of each classifier is known and clear. Moreover it improves the code maintainability, a terms used to indicate how easy is to maintain the software. This is an important topic because an high maintainable code improves the general performance of the system in many ways, for instance allowing to fix bugs easier and improving the usability.

4.1.1 AbstractClassifier

The AbstractClassifier is an abstract class representing the interface implemented by all the other classifier objects. An abstract class is often used as a basis for creating specific objects that conform to its protocol, i.e. the set of operations it supports.

In object-oriented programming (OOP) languages (such as Python), abstract classes are useful when creating hierarchies of classes because they make it possible to specify an invariant level of functionalities in some methods, but leave the implementation of other methods until a specific implementation of that class is needed. In this context, abstract classes represent the root of the hierarchy and programmers use inheritance to define child classes that implement the specific functionalities.

An abstract class has at least one abstract method, that does not contain any code but defines the return type, the number and the type of the parameter required by the method itself. This is the reason why an abstract class cannot be instantiated directly. When a derived class is created, it must implement all the abstract methods following the constraints established in the parent class.

AbstractClassifier exposes the following interface, as illustrated in the class diagram in figure 4.1:

- **from_config**: a python *classmethod*. It creates an instance of the current class given the configuration file.
- **add**: this method receives a parameter value, extracts the feature specific to the current classifier and stores it as a training sample.
- **fit**: this method uses the training features collected through the add method and fits the classifier, meaning that it builds the model corresponding to the current classifier and it defines an anomaly threshold.
- **predict**: this method receives a value and determines if it is anomalous or not.
- **from_dict**: this method restores a JSON serialized version of the current classifier (represented by a python dictionary) into the actual object.
- **to_dict**: this method transforms the current object into a JSON serialized object (i.e. a python dictionary).
- **is_ready_for_fitting**: a python *property* that tells if the classifier is ready to be fitted.
- **is_fitted**:a python *property* that tells if the classifier is fitted or not, i.e. if the classifier is still learning the model or if it is giving predictions.

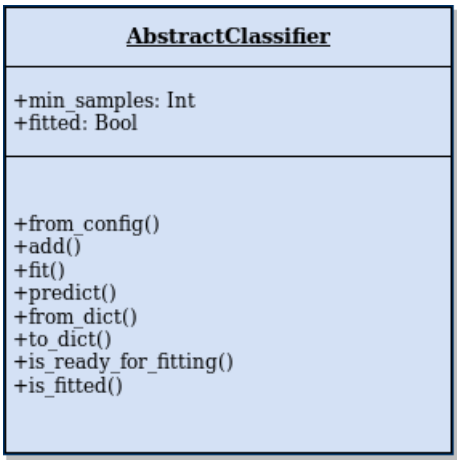

Figure 4.1: AbstractClassifier class diagram.

Based on the above description it is possible to infer the general behavior of a certain classifier. Initially an instance is created and the required parameters are taken from a configuration file calling *from_config*. Then the *learning* phase begins and it starts building the related model. As the web requests are processed by HTTPServerPlugin, the classifier receives attribute values and learns the regular behavior for that parameter collecting feature values through the add method.

Next, when the classifier will be ready to consolidate the model (i.e. *is_ready_for_fitting* returns True), the *fit* method will be invoked and, given the collected metrics, it will sets up an anomaly threshold for future tests values. This is the equivalent of the *validation* phase as described in [1].

Finally, when the model is fitted, the classifier switches in *testing* mode and, thanks to the *predict* method, it will start flagging the received values as benign or malicious, based on the anomaly threshold established during the fitting process. Moreover, periodically the classifier will be serialized on disk and its state later restored, respectively invoking the *to_dict* and *from_dict* methods. This is done because the classifiers are stored inside HTTPServerContext, a LLAnta context class that is regularly serialized on disk for several reasons (as explained in section 2.4.2).

In the following sections I am going to describe in details each concrete class, explaining how each child classifier implements the above abstract methods with its custom logic.

4.1.2 LengthClassifier

The objective of this model is to approximate the actual but unknown distribution of the parameter lengths and detect instances that significantly deviate from the observed normal behavior. To achieve this goal I considered two different metrics: the *Chebyshev's Inequality* and a variant of the well-know *zscore* statistical tool, known as robust (or modified) zscore.

Concerning the first technique, as stated by Kruegel et al. in [1], one drawback is that it computes a weak bound. Nevertheless, according to the authors, this defect can actually be turned into an advantage because in this way the model flags as anomalous only significant outliers.

However, analyzing samples of real traffic, I noticed that the bound was too loose. What happened was that the probability value returned by the *Chebyshev's Inequality* was almost always equal to 1 (or even higher and therefore it saturated to a value of 1). As a consequence, since the anomaly threshold was computed as the highest probability in the training set less a small percentage, the sub-classifier ended up to be very sensible and it was producing several false positives.

Moreover, the *Chebyshev's Inequality* is based on the mean and variance measures. As described by Rousseeuw in [3], these metrics are not robust in presence of outliers in the training set, that is the exactly working scenario of the HTTPServer anomaly detector).

Rousseeuw presents an analysis of the most common statistical tools, the effect of outliers on those and outlines a method to build a robust outlier detector. In order to investigate how robust an estimator is, he considers the *breakdown point*, that is the smallest fraction of observations that have to be replaced to make the estimator unbounded. In this definition one can choose which observations are replaced, as well as the magnitude of the outliers, in the least favourable way [3].

An outlier identifier is made of two components: a location estimator T, that measures the general position of the data, and a scale estimator S, that gives information about the spread of the data. Examples of a weak location estimator and a weak scale estimator are the sample mean and the standard deviation respectively. Concerning the first metric, the breakdown point applied to a sample $\{x_1, x_2, ..., x_n\}$ of n observations, is equal to $\frac{1}{n}$ because it is sufficient to replace a single observation by a large value. For the standard deviation, the breakdown point is also $\frac{1}{n}$, meaning that a single outlier can lead to the explosion of this estimator.

In opposition to these weak metrics, Rousseeuw introduces the sample median as local estimator and the median absolute deviation (MAD) as scale estimator. In both cases the breakdown point is in fact 50%, a much higher value than $\frac{1}{n}$, if there are a reasonable number of samples.

The following two examples can help to better understand what the breakdown point measures and how much more robust the sample median is with respect to the mean:

- Given the set of five observations x:

$$x = \{6.22, 6.5, 6.12, 6.36, 6.40\} \tag{4.1}$$

  The sample mean μ is equal to 6.32 while the sample median md is equal to 6.36. Notice that these two values are not the same but they are very close.

- Consider now the same set of five observations where one element (the last one) has been replaced and it is strongly different from the others:

$$x = \{6.22, 6.5, 6.12, 6.36, 64\} \tag{4.2}$$

  In this case, the sample mean is equal to 17.84, while the sample median did not change (always 6.36). This shows how much the average is sensitive to outliers and the same happens with the two scale estimators, the standard deviation (weak) and the median absolute deviation (robust).

Given a location estimator and a scale estimator, the author derives an outlier identifier, the *z-score*:

$$z_i = \frac{x_i - T}{S} \tag{4.3}$$

where z_i is the *z-score* for sample x_i. The classic *z-score* uses the mean as location estimator and the standard deviation as scale estimator. However, Rousseeuw proposes a robust version using the sample median as T and the median absolute deviation as S.

Once the outlier identifier is defined, a threshold needs to be set up in order to determine if a value is an outlier or not. The author proposes a cut-off value of 2.5. The choice is arbitrary but the probability that a zscore value $|z_i|$ is greater than 2.5 is very low.

For these reasons, I decided to change from the *Chebyshev's Inequality* to this more robust metric. In particular, Iglewicz and Hoaglin in [4] and IBM in [14] propose two variants of Rousseeuw's robust *z-score*:

- The first ones add a scale factor to the basic formula:

$$z_i = \frac{0.645 * (x_i - md)}{MAD} \tag{4.4}$$

  Moreover, they suggest to set up the anomaly threshold to 3.5, rather than the Rousseeuw's recommended value of 2.5.

- IBM, instead, not only introduces a scale factor, but also proposes an alternative formula if the median absolute deviation is equal to zero:

$$z_i = \begin{cases} \frac{x_i - md}{1.486 * MAD} & \text{if MAD} > 0 \\ \frac{x_i - md}{1.253314 * MeanAD} & \text{if MAD} = 0 \end{cases} \tag{4.5}$$

  where MeanAD is the mean absolute deviation, used to replace MAD.

After some tests I ended up using the metric presented by IBM, with one addition: if also MeanAD is equal to 0, the *zscore* simply measures the number of times the length of the value under analysis is greater than the collected median. Therefore, the complete formula becomes:

$$z_i = \begin{cases} \frac{x_i - md}{1.486 * MAD} & \text{if MAD} > 0 \\ \frac{x_i - md}{1.253314 * MeanAD} & \text{if MAD} = 0 \\ \frac{x_i - md}{md} & \text{if MeanAD} = 0 \end{cases} \tag{4.6}$$

Once delineated the statistical tool used by the LengthClassifier, its general workflow is the following:

1. An instance of the sub-classifier is created given the configuration file and the *min_samples* parameter, defining the minimum number of learning samples required to allow the sub-classifier to fit, is set.

2. The sub-classifier starts its training phase. At every invocation of the *add* method, the length of the training value is recorded. Notice that I do not use a list to store the attribute lengths (that would be an obvious choice), but I rather manage a python *collection.Counter*.
   This data structure allows me to count the number of elements with a certain length. In this way, if for example the attribute is a constant (i.e. the length is fixed), I avoid storing the same value multiple times for every sample but I just record the number of occurrences. This is an optimization expedient that allows to save a lot of memory at run-time, especially if there are many training samples with the same length.

3. When the sub-classifier is ready to be fitted, i.e. it has received a number of samples that is greater than *min_samples*, *is_ready_for_fitting* returns *True* and the *fit* method is invoked. During the fitting process, the three discussed metrics (median, median absolute deviation and mean absolute deviation) are computed using the training samples collected so far. Then the *zscore* of each training value is computed and the anomaly threshold is defined by multiplying the highest *zscore* by a predefined factor. This product is done in order to create some margin starting from the most anomalous training sample. Finally the *collection.Counter* containing the training lengths is emptied, to both save memory and to allow for future retraining.
   Notice that the sub-classifier can be retrained: in this scenario, when the fitting method is invoked for the second time, the consolidate metrics are the average between the current and the previous ones.

4. Once fitted, the sub-classifier automatically starts returning prediction for the input value, instead of collecting its length for training purposes. When the testing value is received, the model computes its *zscore*. If it is higher than the established threshold, the value is considered as anomalous and the sub-classifier returns to the caller the state that led to this decision. The state includes the computed *zscore*, the threshold, the median of the lengths seen by the model during training phase and the length of the anomalous value. This information about the anomaly will be used later on to show information in the customer user interface. Otherwise, if the *zscore* is below the threshold the model returns nothing.

The sub-classifier includes also the *to_dict* and *from_dict* methods to allow the sub-classifier serialization and deserialization on disk, and the re-implementation of the *__add__* and *__len__* magic methods. More in details, when two LengthClassifiers are added, the training samples of the two are merged together only if neither of the two sub-classifiers is already fitted, otherwise an error is returned. The length of the sub-classifier, instead, is defined as the number of samples in the *_attribute_length* variable.

The class diagram of the LengthClassifier model is shown in figure 4.2.

4.1.3 DistributionClassifier

The DistributionClassifier model captures the concept of a regular query parameter by looking at its character distribution [1]. It is a re-implementation of the Attribute Character Distribution Model presented in 3.2.1, with some major improvements.

First of all the analysis is focused on the byte frequencies without losing the connection between value and corresponding character. This means that, unlike in [1], the byte distributions are not sorted in descending order. Moreover, even if the metric used is the same (the Pearson's χ^2-test for the goodness of fit), the way in which the observed frequencies O and the expected frequencies E are created from the character distributions is different and therefore the comparison is different from what happens in [1].

More in details, Kruegel et al. divide the function domain of a character distribution into the following six intervals:

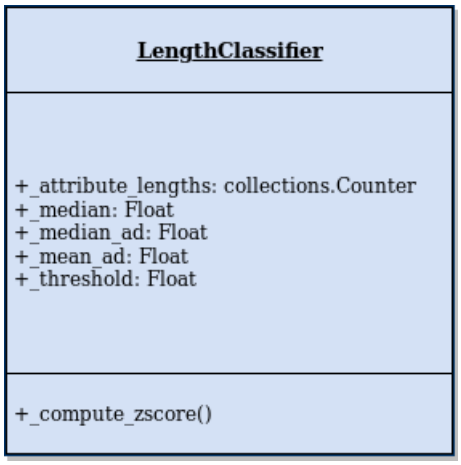

Figure 4.2: LengthClassifier class diagram.

1. [0]
2. [1, 3]
3. [4, 6]
4. [7, 11]
5. [12, 15]
6. [16, 255]

This choice is completely arbitrary but reflects the fact that lower indices in the character distribution contain an higher number of occurrences, because of the sorting in descending order.

In the DistributionClassifier, a character distribution is also split into six intervals, but the choice is not arbitrary. Leveraging on the matching between byte numeric value and corresponding character, the byte occurrences are grouped in such a way that characters belonging to the same class are all together. The six defined classes are:

1. ASCII control characters.
2. ASCII digits.
3. ASCII lower case alphabetic characters.
4. ASCII upper case alphabetic characters.
5. ASCII special characters.
6. ASCII extended characters.

This is a more reasonable choice that follows a specific pattern. In this way, the χ^2-test compares the structures of the expected and observed distributions, meaning that it analyzes the type of characters that constitute a parameter value. Furthermore, since the limitation of this technique is that the χ^2-test is very unreliable when the bins have low values (as stated in section 3.2.1), I decided to set the starting value of each bucket in both E and O to 5, which is the recommended value in literature to make the test work correctly. Of course, this choice is arbitrary but, since the goal is to detect the outliers in a distribution of values which are very far from the regular samples, it should not affect the analysis negatively.

The DistributionClassifier works as follows:

1. A new instance of the sub-classifier is created given the configuration file.

2. The sub-classifier starts its training phase. At every invocation of the *add* method, the absolute character distribution of the value is extracted by counting the occurrences of each byte. As for the LengthClassifier, the character distributions are not stored in a list but in a python *collection.Counter*, thus allowing to save memory at run time.

3. When the sub-classifier is ready to be fitted, the *fit* method is invoked. During the fitting process, I divide each collected distribution by its length (deriving the relative byte frequency for each training value) and I store the average of the so obtained distributions, that represent what in [1] is referred to as Idealized Character Distribution (ICD).
   Then I compute the χ^2 probability by comparing the ICD with every training distribution and I set the threshold by dividing the worst probability (i.e. the lowest) by a predefined factor. Finally, I empty the python *collection.Counter* to allow for future re-training like for the LengthClassifier.
   Notice that I do not compute directly the relative byte occurrences in the training phase because I need to keep the information of the length for each value. If I compute the relative character distribution in the training phase, I would not be able to calculate the threshold during the fitting phase, because the length of each training value would be missing and it would be impossible to apply the test.

4. Once fitted, the sub-classifier starts working in testing mode and it returns predictions for the input values. When a testing value is received, the model extracts its absolute character distribution and group the values according to the classification above. These buckets represent the observed frequencies O. The expected frequency E, instead, are built starting from the average distribution. The values are again grouped following the defined pattern and, in addition, each bucket is multiplied times the length of the testing input. In this way, from a relative measure, we obtain the expected number of occurrencies for each bucket.
   Then, the χ^2-test is applied. If the probability returned by the test is lower than the threshold, the sub-classifier flags the value as anomalous and returns the state that led to the prediction. The state just includes the probability of the testing value and the threshold. Otherwise, everything is considered fine and no alert is reported.

Like the LengthClassifier, this sub-classifier has the capability to serialize/deserialize its state to/from disk, it defines the concept of length and it supports the use of the + operator.

The class diagram is shown in figure 4.3.

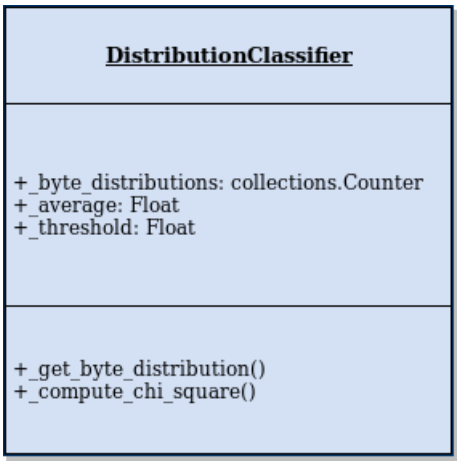

Figure 4.3: DistributionClassifier class diagram.

4.1.4 DataTypeClassifier

The DataTypeClassifier analyzes the input traffic and tries to determine the data type(s) of a given attribute based on the observed values. Currently are supported 8 different data types, each of them with a custom check function:

- Base64.
- Email.
- MD5.
- SHA1.
- SHA256.
- URL.
- UUID.
- Hostname.

These data types have been chosen because are commonly exchanged over the Internet. For instance, it is frequent to send a file along with its hash, or to use an UUID to identify a certain resource.

The possible data types of a certain value are guessed providing the value in input to the *_compute_data_types* function, that runs every check function against the value. Each check function will return the corresponding data type if there is a match and nothing otherwise. Moreover, if there is a match and the particular data type is a kind of encoding (for instance in the case of *Base64* encoding), the value is decoded and the check functions are recursively applied to the decoded value. This process is repeated until a maximum level of recursion depth is not reached or no check function matches the current value. At the end, *_compute_data_types* returns a tuple with the matching data types, which can also be empty if the value does not belong to one of the supported categories.

The supported data types and corresponding check functions are the followings:

- **Base64→_check_base64**: Base64 is a form of binary-to-text encoding that converts binary data in an ASCII string format. It is generally used to transfer content-based messages over the Internet sent over media that are designed to deal with textual data, like emails. This is to ensure that the data remain intact without modification during transport. It is also very common as a form of light obfuscation or to encode even plain text data, because there may be some systems (usually quite old) that can handle a limited character-set only. For instance, from the traffic I analyzed, some web servers where designed to receive encoded urls or hostnames.

  The encoding works by dividing every three bits of binary data into six bit units. The newly created data is represented in a 64-radix numeral system and as seven-bit ASCII text. Because each bit is divided into two bits, the converted data is 33 percent, or one-third, larger than the original data. The increase may be larger if the encoded data is small. For example, the string "a" with length equal to 1 gets encoded to "YQ==" with length equal to 4, with a 300% increase.

  To assess if a value is Base64 encoded, the *_check_base64* method makes use of the *base64* python library, that provides support for the encoding and decoding operations. Unfortunately there is no way to determine if a string that is compliant to the Base64 encoding standard it is actually Base64 encoded or not. For instance, the string "aaaa" is a valid Base64 input and can be decoded, but it is unlikely that the desired value is the decoded one. Therefore, the method first tries to decode the value as Base64. If the string cannot be decoded, for sure it is not a Base64 value and no match is returned. Otherwise, if the decoding is successful, the decoded value is accepted only if every obtained character is ASCII printable. In this case, the method returns a match and also the decoded value, in order to apply the checking functions recursively.

- **Email→_check_email**: The checker function uses a regular expression to look for a match.
- **MD5→_check_md5**: MD5 (Message-Digest algorithm) is a widely used hash function that transforms an arbitrary input into a 128-bit output, which can be represented as 32 hexadecimal values. It was originally designed for use as a secure cryptographic hash algorithm for authenticating digital signatures, but it has been later deprecated because it suffers from many vulnerabilities. Nowadays it is used to verify data integrity and detect unintentional data corruption, for instance due to transmission errors.
  The checker function makes use of a simple regular expression.
- **SHA1→_check_sha1**: SHA stands for Secure Hashing Algorithm. SHA-1 is the first member of this family and it produces a digest of 160 bit (40 hexadecimal digits). It has been the main algorithm used by SSL for digital signatures for many years, until it was replaced by its successor SHA2 in 2016.
  Once again, a SHA-1 hash value is identified using a regular expression.
- **SHA256→_check_sha256**: SHA256 (or SHA2) is a variant of the SHA hash family that produces an output of 256-bit (64 hexadecimal values). As all the other hashes, the data type is inferred using a regular expression.
- **Url→_check_url**: It is quite hard to properly identify urls, not because they do not have a well defined structure, but because often web servers are tolerant and accept inputs even if they do not follow exactly the standard. The goal is therefore to be able to accept the most number of legitimate variants and reducing the number of false positive while avoiding to accept everything.
  Firstly I tried to use a regular expression, but I soon realized it was too generic and it was matching practically every input. Thus, I decided to separate the identification of an hostname only from the URL data type (that of course can contain an hostname). In this way I was able to make stricter the URL pattern recognition.
  More in details, I used the *urlparse* python library. This module is able to parse an URL string into its single components [15] (corresponding to the ones presented in section 3.1:
  - **scheme**: URL scheme specifier.
  - **netloc**: network location part.
  - **path**: hierarchical path.
  - **params**: parameters for last path element.
  - **query**: query component.
  - **fragment**: fragment identifier.

  If *urlparse* is not able to identify a certain component the resulting value will be an empty string. Hence, after I parse the input value, I consider it an URL if either a scheme or a netloc are present.
- **UUID→_check_uuid**:An UUID is 128-bit (16 bytes) number used to identify information in computer systems. Although the probability that a duplicate exists is not zero, it is close enough to zero to be negligible and it is used for practical purposes [9]. An UUID is usually displayed in 5 groups of hexadecimal digits separated by hyphens, in the form 8-4-4-4-12. For example, the string *123e4567-e89b-12d3-a456-426655440000* is a valid UUID.

  There exists many versions and variants that have been defined over the years and these are specified by some bits in the UUID itself.

  I used the *uuid* python module to identify a string as a valid UUID. This module generates an UUID object from the corresponding string representation and raises an exception if the provided input is not well-formatted. Therefore, in order to check the data type in input to the sub-classifier, I simply try to create the UUID object and, if an exception is raised, this means the input string was not a valid UUID.
- **Hostname→_check_hostname**: For the hostname recognition I used a regular expression that matches any hostname, including IP addresses or local hostnames that can have no dots.

The DataTypeClassifier works as follows:

1. A new instance of the sub-classifier is created given the configuration file.
2. During training phase, at every invocation of the *add* method, the sub-classifier tries to determine the data type(s) of the input value according to procedure described above. If there is a match with at least one type, a counter is increased to take track of how many times the value corresponds to a certain category.
3. When a sufficient amount of samples are processed, the sub-classifier is ready to be fitted and the *fit* method is invoked. During the fitting phase, if one or more data types have been observed for a number of times equal to the 80% of the training samples, then the attribute is assigned to that matching pattern. Since re-training is allowed, the patterns are stored in a set and, every time the sub-classifier is fitted, the new matching data types are added to those already present in the set.
4. Once fitted, the sub-classifier switches to testing mode and it starts giving predictions for the input values. When a value is received, the model infers its data type(s) and, if it matches with the established pattern it does nothing, otherwise it flags the value as anomalous and return its state. The state simply contains the computed data type(s) for the current value and the expected ones.

As for the other classifiers, the DataTypeClassifier offers the capability to be serialized and de-serialized to and from disk. Moreover it redefines the + operator and it defines the concept of length.

The class diagram is shown in figure 4.4

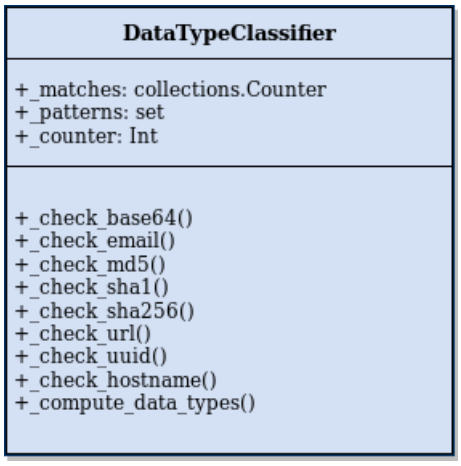

Figure 4.4: DataTypeClassifier class diagram.

4.1.5 AttributeModel

The AttributeModel class is a classifier much more complex than the ones presented so far. It can be considered as an aggregation stage for the other sub-classifiers and it is the model that represent the behavior of a regular attribute of an HTTP GET request.

An AttributeModel combines together all the other classifiers and, using the techniques implemented by the sub-classifiers, it is able to understand if a certain value is anomalous or not. It also builds and applies several other heuristics in order to improve the analysis and avoid consuming too much memory at runtime and space on disk.

First of all, the AttributeModel implements the same interface as its sub-classifiers (i.e. it inherits from the AbstractClassifier class) even if it is an higher granularity level with respect to the others. Therefore, the basic workflow is the same:

1. There is a training phase during which the model collects samples and learn the regular behavior of the attribute.
2. Then follows a fitting phase to fit all the sub-classifiers.
3. Finally the classifier switches to testing mode and start making predictions on the input values.

However, on top of this, the AttributeModel adds two extra functionalities:

1. The model learns what is a legitimate value for the given attribute for each different client sending HTTP requests to the target web server containing that attribute. Of course, learning a model based on the same traffic on which predictions are made is sub-optimal because it is impossible to know in advance if the traffic used to learn the model is benign or it contains some malicious input. The latter scenario is very likely to happen, especially if the web server is exposed to the public Internet, where there are plenty of malicious sources.

   Therefore, one could argue that this kind of learning is not suitable for the HTTPServer working scenario. However, the learning per client helps to face this problem and, at the same time, overcomes the limitations of both [1] and [2]. In fact, the learning per client allows to consider the traffic coming from each client as all benign. In this way, each client has its own LengthClassifer, DistributionClassifier and DataTypeClassifier and each of them separately builds the concept of what are the legitimate values for that attribute.

   In a second time, when the AttributeModel switches to testing phase, each client will be asked to evaluate the input and give its vote (malicious/benign). Using a majority vote voting system, the AttributeModel will then consider the value as truly malicious or benign.

   This whole system works under one **assumption**: I assume that the majority of the clients sending traffic to the target web server are benign. Since usually legitimate users are much more frequent than attackers, this should be reasonable enough. In this way, even if some clients were malicious and thus they do not consider as anomalous another malicious input, the hope is that there will be enough benign clients recognizing the attack, therefore obtaining an overall malicious vote.

   By implementing the learning per client and the concept of majority vote, we then solve the problem of a contaminated data set for the training phase and we defeat the two main limitations of the previous works in literature.

   In fact, on one hand, Kruegel et al. in [1] make use of two different data sets for training and validation and, in order for the system to work properly, they have to be clean from any attack. This means that the inputs need to be collected in advance and undergo a cleaning process if some malicious input is present. However, training and validation done in such a way can cause the metrics to become obsolete very soon and therefore re-training (involving the whole cleaning procedure) is required quite often. HTTPServer instead, learns, validates and tests processing only real traffic from the customer network.

   On the other hand, Krueger et al. in [2] implemented a system that is able to learn the anomaly detectors using even contaminated data sets, but the setup of the system is not fully automatic. In fact, a system administrator is needed to monitor the learning and validation phases and to adjust some parameters. HTTPServer, instead, does not require any human intervention to prepare the system: all the necessary parameters are read from a configuration file and there is not a pre-processing phase to make the inputs ready to be analysed.

2. Second, like the other sub-classifiers, the AttributeModel is periodically serialized on disk. The AttributeModel stores a quite consistent amount of information and one object is created for every parameter seen in the HTTP requests sent to the customer web server. As a consequence, the quantity of memory used by HTTPServer at runtime and the chunk of disk used when it gets serialized increases deeply as time passes.

   In order to cope with this problem, an AttributeModel has a maximum number of clients it can accept into the voting system. If the limit has been reached and a value from a client

never seen before is received, the model implements a LRU (Least Recently Used) policy: the information related to the client that did not send any input for the largest amount of time is deleted and the new client is inserted in the model.

Moreover, the AttributeModel implements the concept of *expiration* of a classifier: a classifier is defined as expired if it has not received any input in a certain amount of time (for instance one day) and it is not fitted. This cleaning process works like a garbage collection mechanism.

Periodically, the AttributeModel object checks if there are any expired classifiers belonging to the observed clients and it removes them from the corresponding dictionary. However, instead of throwing away all the information collected so far, each classifier is merged into a corresponding *Default* sub-classifier, which is not associated to any client. This is why every sub-classifier overloads the + operator, allowing to perform the sum in a very easy and straightforward way.

The idea is that, if in a certain amount of time a sub-classifier does not receive enough samples to be fitted, probably it is not essential for the voting system and we can free memory and space on disk. Anyway, the information is not all lost because the sub-classifier is merged into the corresponding default one (until the default sub-classifier is not fitted, otherwise nothing happens).

Therefore, the default sub-classifier of a sub-classifier collects all the inputs from unfitted sub-classifiers belonging to different clients. As a consequence, a default sub-classifier can contain any kind of input and we cannot assume that the samples are all benign like we do when we consider the traffic separately for each client. This is the reason why the predictions made by a default sub-classifier needs to be treated differently and have a different weight compared to the others.

Since each sub-classifier can define the minimum number of samples it needs to be ready to fit, it may happen that, for a certain client, some sub-classifiers expire and some not. In the borderline case in which all the sub-classifiers of a client are expired and removed, the client itself is considered expired and removed from the corresponding dictionary.

In conclusion, the expiration process has two main advantages. On one hand it avoids consuming to much memory and space on disk for not very valuable information. On the other hand, the information is not completely lost and can still be used to improve the analysis, even if it is less reliable.

The workflow is the following:

1. A new instance of the classifier is created given the configuration file. In this phase, the AttributeModel also creates one instance of each sub-classifier that is the default sub-classifier for that kind.

2. When the *add* method is invoked, the AttributeModel first checks if the client (i.e. the source IP address) has already been seen before. If not and the maximum number of clients has not been reached, it creates an instance of each sub-classifier and it assigns them to that client. If the maximum number of clients is already registered for the classifier, the LRU policy is applied as explained above. Otherwise, if the client was already present, it is retrieved from the pool, the timestamp of its last time activity is updated (it is needed for the expiration process) and the value is used to train each sub-classifier if they are not yet fitted (invoking again their *add* method).

3. The fit method receives as parameter the list of the client IP addresses that have one or more sub-classifiers ready to be fitted. For every client in the list, the AttributeModel checks which of its sub-classifiers can be fitted (invoking *is_ready_for_fitting*) and it subsequently calls the *fit* method. The AttributeModel has also the capability to force the fitting of a sub-classifier even if it is not ready and this setting is enabled passing to the method an optional boolean value.

   Once the classifier are fitted, the AttributeModel also run the cleaning process as described above.

4. Finally, the *predict* method is used to determine if a certain value for the related attribute is anomalous or not. As described above, the prediction makes use of a voting system. To achieve this goal, the AttributeModel collects the predictions of all the sub-classifiers of all clients. Then, if there are enough samples that gave an opinion (benign/malicious) about the testing value, the model counts how many clients considered the value as malicious and, if this number is higher than the 70% of the total voting clients, the value is flagged as truly malicious.

   The vote of a client consists of a combination of the predictions of its sub-classifiers. For the time being, all the sub-classifiers are treated equally and therefore it is enough that one of them gives a negative vote to make the client answering malicious.

   Once a value is considered malicious, the AttributeModel collects the state of all the classifiers giving a negative prediction. This step is extremely important because it provides the upstream components (HTTPServerPlugin and HTTPServerDetector) with the information needed to understand what led the AttributeModel to take this decision. The state will undergo a further processing stage and it will finally be shown in the customer user interface (UI) to help them have more context about what is happening in their network.

One important thing to notice is that, unlike the sub-classifiers, the AttributeModel never ends the training phase and keep learning all the time while giving prediction once ready. In fact, while the other classifiers do not receive values for training once they are fitted, the AttributeModel keeps adding clients until a maximum number is reached and, even after that, the model does not stop adding clients and it updates its metrics by replacing the oldest (i.e. less relevant) client.

In this way the AttributeModel is able to improve the accuracy of its predictions overtime. By gradually increasing the number of clients that contribute to the voting system, it is more likely to detect an anomalous value, given that the assumption that the majority of the clients contacting the web server is benign holds.

Since there is not the concept of default classifier for the AttributeModel, it does not redefine the + operator, while the length is defined as the sum of the lengths of the sub-classifiers of all clients.

The class diagram is shown in figure 4.5.

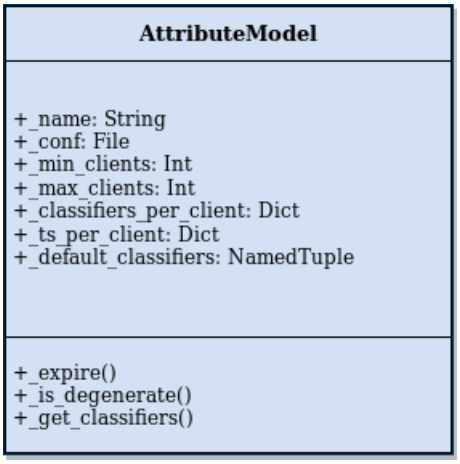

Figure 4.5: AttributeModel class diagram.

This completes the analysis of the classifiers of the HTTPServer anomaly detector. The full class diagram with all the relationships among the components is illustrated in figure 4.6.

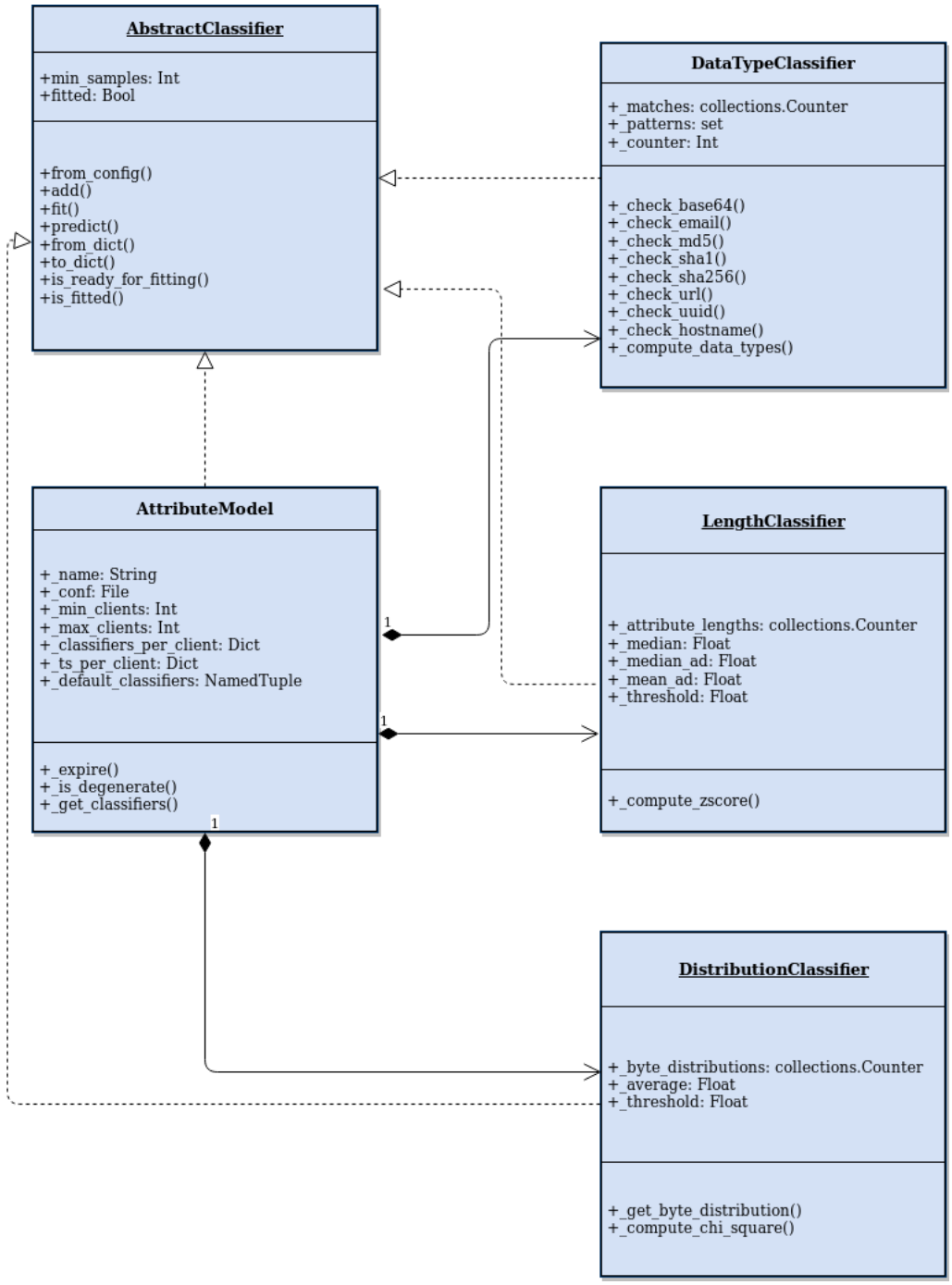

Figure 4.6: Classifiers class diagram.

4.2 HTTPServerPlugin

HTTPServerPlugin is the main component of the whole system and it is a LLAnta plugin as presented in section 2.4.2. It is scheduled any time a new web request is received by LLAnta and it performs the core processing, in order to ease the work done later by the LLAnta detector, represented by the HTTPServerDetector class.

When an instance of the plugin is created, it retrieves its attributes from a configuration file. The same configuration file contains the settings for the AttributeModel class and all the sub-classifiers and, therefore, it is stored by the plugin to pass it to every instance of the AttributeModel class. Notice that an AttributeModel needs to store the configuration file to be able to pass it to the sub-classifiers and that each AttributeModel is stored in the HTTPServerContext class, which is periodically serialized on disk.

After the first deployment I noticed that this design choice has a consistent drawback. In fact I did not take into account that there is a single configuration file for every plugin and detector in LLAnta. Even if the size itself it is not big (it is currently around 6 KB), when replicated for each attribute of each HTTP request, it was resulting in a huge waste and it was slowing down the performance.

To solve this problem, when a new instance of the HTTPServerPlugin class is created, instead of storing the full configuration file, the plugin just keeps the options needed by the AttributeModel and the sub-classifiers. The file will still be replicated for every AttributeModel but its size is irrelevant, as it only contains very few parameters.

The HTTPServerPlugin receives web requests in the form of IDSURL objects, whose class diagram is shown in figure 4.7:

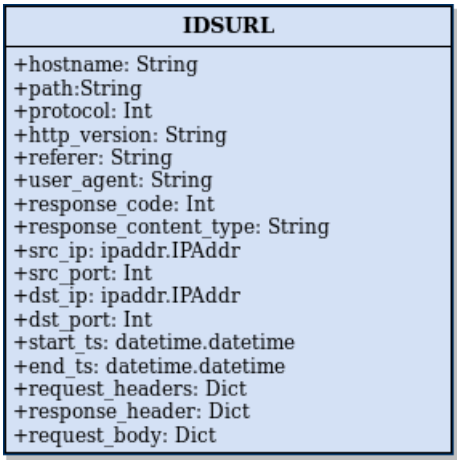

Figure 4.7: IDSURL class diagram.

When the plugin is scheduled it processes the web request only if its destination IP address belongs to the customer network, named *managed network*. LLAnta automatically learns the boundaries of the customer network, that can differ from the actual reality.

The basic architecture for a network in LLAnta is based on a python *NetworkGraph*. A NetworkGraph is a direct graph based on networkx.DiGraph. Each node of the composed graph is an IP address. A directed edge between two nodes exists if there has been a connection between the two IP addresses and the direction is determined according to the source and destination of the connection. Upon receiving an URL (i.e. an IDSURL object) both source IP and destination IP (and their relative edge) are added to the graph.

It is important to notice how each entry in the graph, nodes and edges, have an expiration timeout. Upon insertion, each entry is assigned with the timestamp of the input source that generated that entry. This timestamp is updated each time a new input is received with the same

data (source, destination, connection). The network periodically checks its entries and delete every entry whose timestamp is older than a certain configuration parameter.

Once assessed that the target host of the web request belongs to the protected customer network, the plugin extracts the parameters and corresponding values. These are obtained parsing the URL of the request with the *urlparse* library as for the the DataTypeClassifier. In this scenario, the *query* component is needed because it contains the string with the parameters and values.

Since we deal with malicious inputs, before processing this string to get the singles attribute-value pairs, it needs to be manipulated in order to avoid parsing errors. More in details, I decode any url-encoded = character (represented by the sequence *%3d*) and I encode any *;* character (which is converted into the characters *%3B*). The first step is needed because sometimes malicious inputs encode the = character (used in a query string to associate the attribute name to its value) to hide the attribute-value pair. The second modification, instead, is required because in old times the *;* character was allowed together with & to separate the attribute-value pairs. urlparse supports the *;* character as separator for backward compatibility, but this often results in parsing errors, especially in case of command injection attacks.

When all the attributes with their values are retrieved from the web request, the plugin starts the actual processing. Notice that the same attribute can appear multiple times in a query and it can also have an empty value. In the first case all the values are considered independent and they are handled in sequence. Empty values, instead, are discarded and they do not contribute to the analysis.

HTTPServerPlugin stores the AttibuteModels inside an instance of the HTTPServerContext class, an example of LLAnta context. An AttributeModel gets created for each analyzed parameter, separately for each resource path hosted by the target web server (similarly to what is done in [1]). During an early development stage, instead, only one model was created for a certain parameter name of an HTTP query. However, it is true that how a parameter is used internally can differ from a web application to another one and that a single web server (identified by its IP address) can also host multiple web applications. For instance, a certain URL could expect to receive an attribute "id" as a numerical value only, while another one could treat the same parameter as an alphanumerical value. If one model only is created for this parameter, it would not correctly represent the behavior of the traffic targeting the web server and the analysis would be skewed.

When the plugin processes an attribute, it retrieves the corresponding AttributeModel from the context, which creates a new AttributeModel if the attribute was never seen before or returns the one already present. Then, the plugin both adds the attribute value(s) to the model for training and make a prediction on them. This can seem an erroneous design choice because it seems that the same values are used both for training and testing. However, the AttributeModel class is built in such a way that while training it will not return a prediction on the input (therefore the *predict* will not perform any action). On the other hand, if all the AttributeModel sub-classifiers are already fitted and the input comes from a client already seen before, the *add* method will be ineffective and only the prediction will be done.

If for the current value the prediction cannot be obtained yet or the value was considered benign, the plugin continues with the processing of the following parameters and values of the web request. Otherwise, if the value is deemed as malicious, the plugin stores the evidence associated with the prediction, that corresponds to the state returned by the AttributeModel's *predict* method.

Once all the parameters and corresponding values have been processed by the plugin, if there has been at least one pair considered as malicious, the plugin creates an alert and stores it in the context. An alert is a *llanta.service.objects.Alert* object, whose class diagram is shown in figure 4.8. An Alert is the only way LLAnta has to communicate with the upstream node (the Manager) and report that something is wrong in the customer network. When the Manager receives an Alert, it processes it and it extract from it some crucial information that will be then shown it the customer in the user interface.

A LLAnta Alert contains a lot of data, mainly concentrated in two components:

- **AlertURL**: it contains high-level information about the URL that raised the alert, such as the HTTP method, the protocol, the name and the path.

- **AlertEvidence**: this object includes general information about the alert itself. In particular it carries the state of the anomaly detector that led to the raise of the alert. Concerning the HTTPServer anomaly detector, the alert state consists of the sequence of all the sub-classifiers that considered a given attribute value as malicious.

By design only a LLAnta detector can send the Alerts to the Manager. Therefore, HTTPServerPlugin creates partial Alerts objects (because some fields can be filled only by the detector) and it stores them into the context. They will be later on further processed by HTTPServerDetector and uploaded upstream.

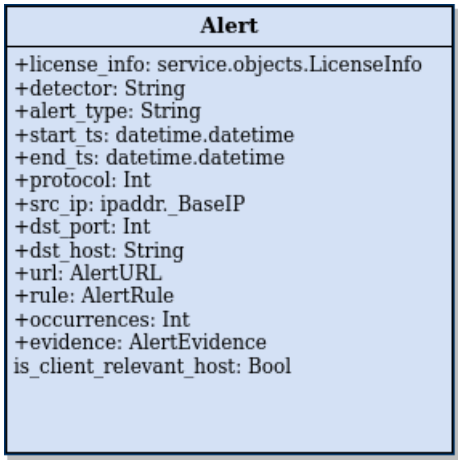

Figure 4.8: Alert class diagram.

Once the alert object is created and saved, the plugin terminates the processing of the input web request and it is ready to work on the next one.

Notice that the plugin does not store anything locally but, instead, both the AttributeModels and the Alerts are stored in the HTTPServerContext. This is done for multiple reasons. First, this information cannot be lost in case of reboot or system failure and, therefore, the context is the right place to put all of it since it gets periodically serialized on stable storage. Moreover, the context is shared between a LLAnta plugin and detector and this comes particularly handy for the HTTPServer anomaly detector because the detector needs to access and process the Alert objects generated by the plugin.

Beside the main processing function, invoked whenever a web request is received by the LLAnta instance, every plugin implements also a *flush* method, which is periodically called every 5 to 10 minutes. This method is usually meant to produce LLAnta Facts, general properties (non malicious) associated to hosts inside a monitored network. For instance, a LLAnta plugin could learn the operating system running on a certain host, or the host role inside the network (such as name server or email server).

However, HTTPServerPlugin is not designed to generate any Fact. Instead, it performs three main actions:

- **Fit the AttributeModels**: every time the method is invoked it accesses the context and goes through all the AttributeModels created so far. For each of them it checks if there are one or more clients that have sub-classifiers ready to be fitted and, if any, it proceeds to fit them.
- **Trigger a cleaning process**: in section 4.1.5 I introduced the cleaning process performed by the AttributeModel class. Eventually if an AttributeModel does not receive enough inputs after a certain amount of time (in the order of days), it will become *degenerated*, i.e. it will be an empty skeleton without any sub-classifier associated to a client. When reaching this condition, the AttributeModels are useless in the sense that they cannot provide any prediction and, since they have not been trained in a long time, it is likely they are not

samples of the attribute population crucial for the analysis. Therefore, the flushing includes the periodic cleaning of all the degenerated AttributeModels from the context.

- **Reset the evidences**: at every flush the plugin discards the evidences collected so far related to all the attribute values deemed malicious. This is done to save memory and it does not affect at all the following analysis. In fact the information stored in the dict containing the evidences is saved in the corresponding Alert objects.

The class diagram of HTTPServerPlugin is shown in figure 4.9.

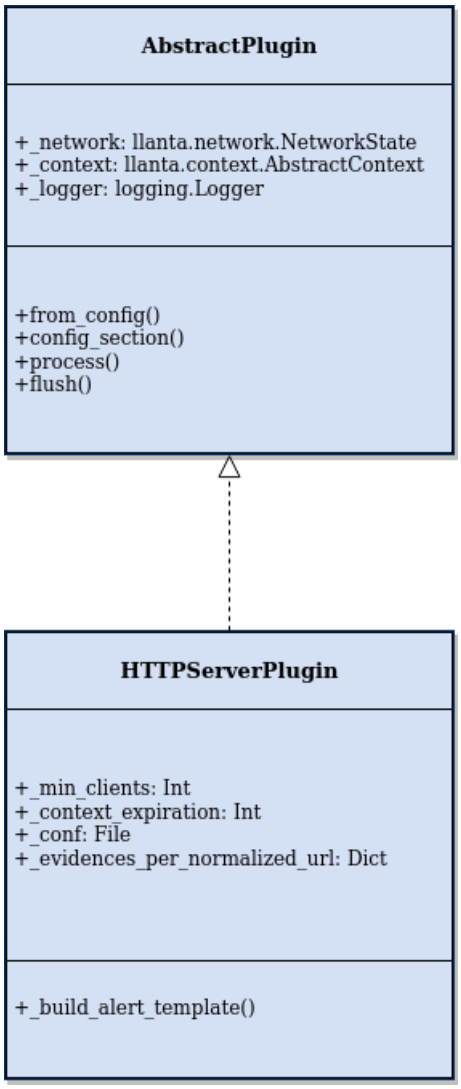

Figure 4.9: HTTPServerPlugin class diagram.

4.3 HTTPServerContext

As explained in the introduction to the company, The LLAnta context is the main way to exchange unstructured information between plugins and detectors. The context is local to a given module and it is shared by the plugin and the detector defined inside the module itself. It is a a key-value data structure, similar to a python dictionary. It therefore allows to store data identified by a unique key and, unlike a classic dictionary, it is possible to set an expiration timeout for the stored information.

The content of the context will be serialized and loaded out of the LLAnta state during flush operations (i.e. periodically every 5 to 10 minutes). As a consequence, the content of the context must be JSON serializable and the classes deriving from the interface represented by the *AbstractContext* class need to implement the two methods *json_encode* and *json_decode*, invoked when serializing and deserializing respectively.

Besides the methods required by the abstract interface, HTTPServerContext also defines other helper methods. In fact, HTTPServerPlugin is designed in such a way that it uses HTTPServerContext as a façade class. The façade pattern is a software design pattern commonly used in object-oriented programming.

Façade pattern is classified as a structural design pattern. This design pattern is all about class and object composition. It is used as a camouflage to cover the complexities of a large system and therefore provides a simple interface to the client. Analogous to a façade in architecture, a façade is an object that serves as a front-facing interface masking more complex underlying code In other words, it is a wrapper class used to hide the implementation details and allows interaction by exposing a set of methods (API) to the client. Moreover, this design pattern, by hiding the implementation details, also improve the readability and usability of of the code.

HTTPServerPlugin makes use of HTTPServerContext to provide a simple interface to the AttributeModel and Alert objects stored in it. Three methods are defined for this purpose:

- **get_attribute_model**: this method is invoked by the plugin every time it needs to provide a training value or to receive a prediction for the value of a certain parameter. More in details, this function returns the AttributeModel corresponding to the parameter under analysis and, if it is the first time a value is received for that attribute, it creates the AttributeModel and provides it to the plugin. Therefore, this is the method used by the plugin to access all the information is generated over time. Remember that an AttributeModel is created for each parameter analyzed by the plugin, separately for each resource path hosted by the web server.
- **clean**: at every periodic flush operation, the plugin attempts to save same memory and space on disk by trying to remove some useless AttributeModels from the context. Therefore the cleaning process performed by this method removes from the context all the degenerated AttributeModels.
- **set_alert**: when the plugin identifies a web request that contains some malicious parameters it creates an Alert object and it stores it in the context. Unlike the AttributeModels objects, the Alert objects are not saved in the context to keep a state of the traffic processed by the plugin and to be resilient to failures (i.e. to be able to restore as much information as possible if the LLAnta instance crashes or gets restarted), but instead this is done to allow HTTPServerDetector to access these data and perform further processing on them.

The complete class diagram for the context is shown in figure 4.10.

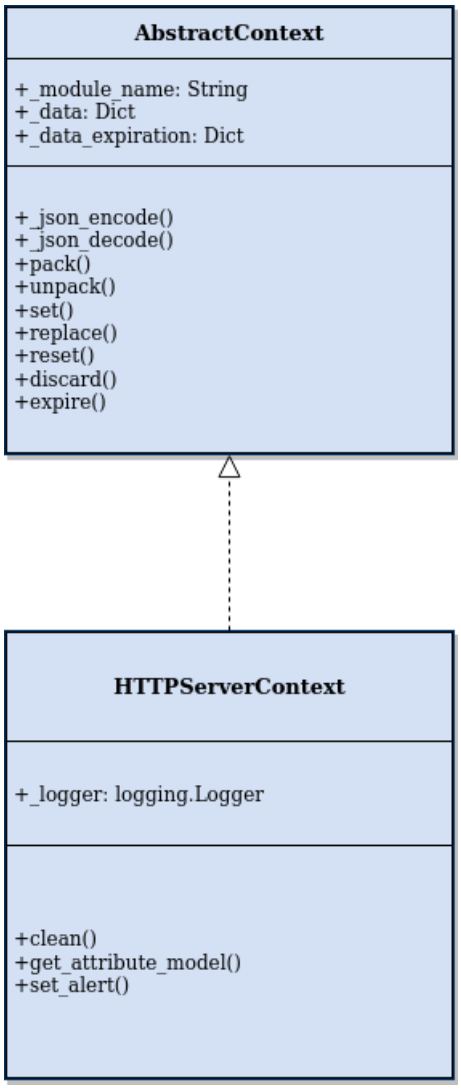

Figure 4.10: HTTPServerContext class diagram.

4.4 HTTPServerDetector

LLAnta detectors are modules that work in cooperation with the plugins. They can perform heavier computation with respect to the plugins and they are in charge of generating anomaly alerts.

Like the plugin, a detector need to register itself to the input family they want to process. The input data type that should trigger the scheduling of the detector is defined by the *TRIGGER_TYPE* function. If the detector is meant to react only to a given type of data (for instane a web request), this method returns the corresponding input type. Otherwise, if the default value is used, it will cause the detector to be triggered by any type of activity.

When the detector receives an input that matches with what is defined in *TRIGGER_TYPE*, *trigger_check* is invoked to decide whether a specific input should actually trigger it or not. This allows a more fine-grained control over what *TRIGGER_TYPE* allows, although the two can be combined together. All the inputs matching a given *TRIGGER_TYPE* will call this method that will decide whether the detector should be triggered based on some specific condition of the input. Concerning HTTPServerDetector, *trigger_check* triggers it if there is at least one alert in the context.

Once the detector receives an input that would trigger the processing logic, *SCHEDULE_DELAY*

defines the amount of seconds llanta should wait before executing the detector on a managed host after the first activity from/to the host is detected. The intuition here is that we want to ensure that, after some activity is detected, we let it run "long enough" to leave a detectable fingerprint in the network state. This value is potentially detector-specific.

The actual detector logic is stored in the *run* method, which is implemented by every class that derives from *AbstractDetector* and it is invoked for the correct input type and after the defined seconds of delay. More in details, the HTTPServerDetector workflow is the following:

1. Retrieve from HTTPServerContext the anomaly alerts reported by HTTPServerPlugin. The alerts in the context are grouped for each resource path targeted by the potential attack and for each client (source IP address) that sent the web request. Along with the alert object, the attribute-value pairs that raised the alert are saved in the context.
2. Apply a filtering mechanism to reduce as much as possible the number of false positives: for each alert related to a given resource path, the detector checks if the same attribute-value pairs that raised that alert were reported by other clients with respect to that resource path. If all the attribute-value couples are reported by at least *_max_clients* different IP addresses, the current alert is considered a false positive and discarded. Moreover, the client IP is inserted into a whitelist.
3. At the end of the filtering process, the detector removes all the remained alerts generated by an IP address belonging to the whitelist. Notice that this can cause the cleaning of alerts related to a different resource path than the one in which the false positive was founded, increasing the overall accuracy.
4. Once the final list of alert objects is obtained, the detector fills the information required to complete them (i.e. the information that the plugin did not have because it is detector-specific). Then all the data stored in the context are removed to save space on disk and memory and the list contaning the alerts is returned to the upstream Manager.

The class diagram of HTTPServerDetector is shown in figure 4.11.

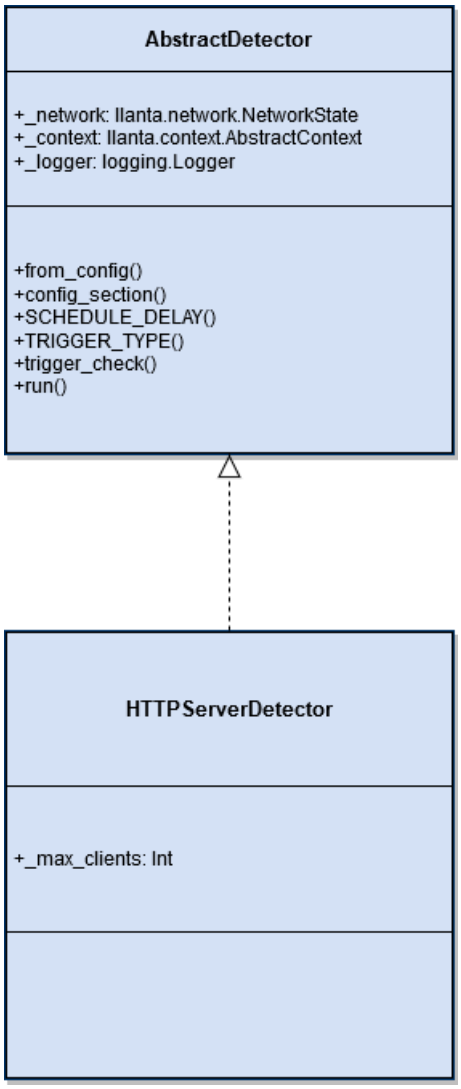

Figure 4.11: HTTPServerDetector class diagram.

Finally, to summarize the structure of the whole system, the complete class diagram of every HTTPServer component with the corresponding interactions among them is shown in figure 4.12.

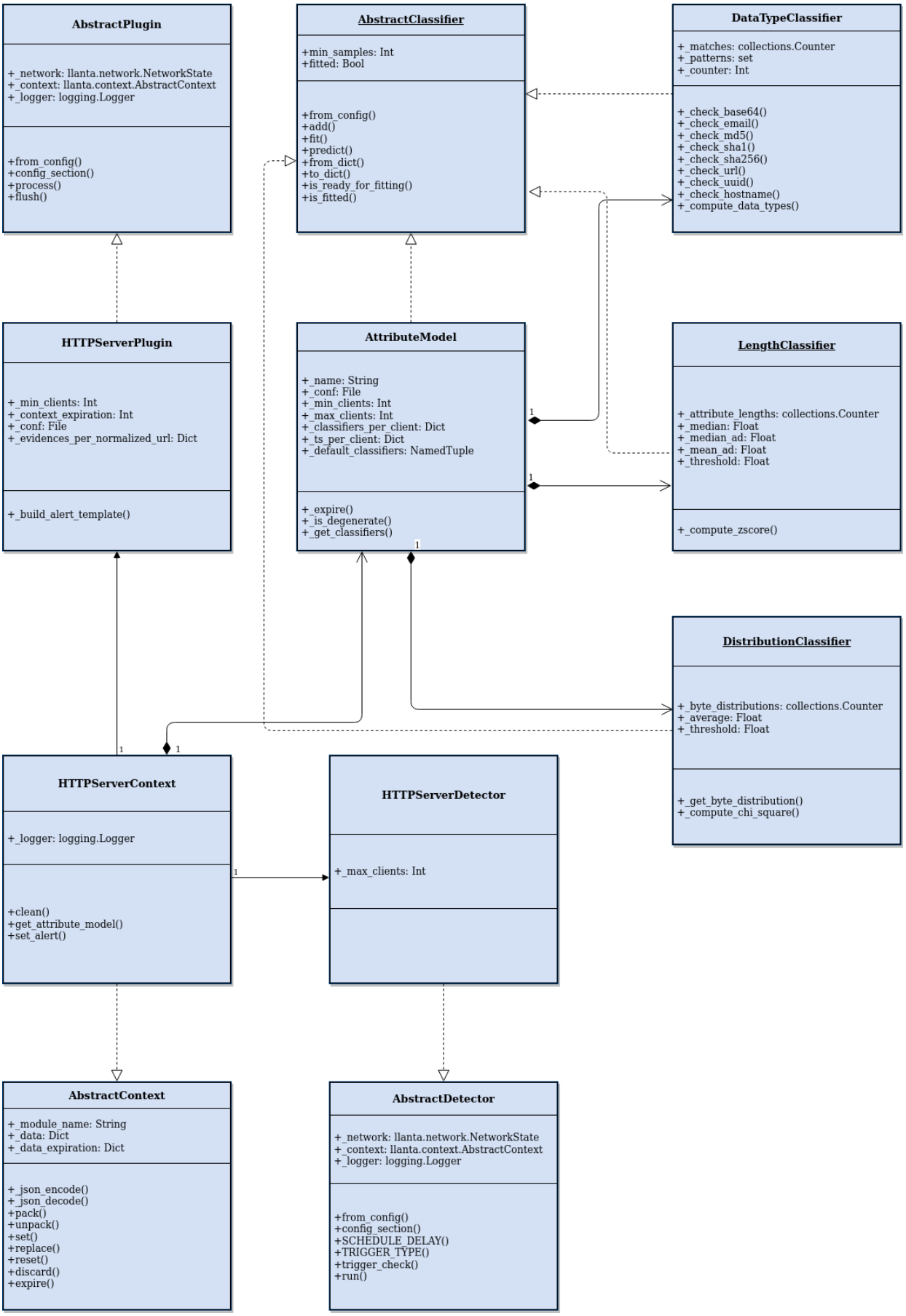

Figure 4.12: HTTPServer anomaly detector class diagram.

4.5 Testing

Testing is a core part of the development process, especially when the code base of a project is very large, like in LLAnta. Testing is a process of executing a program with the aim of finding error. To make the software perform well, it should be error free and, if testing is done successfully, it will find many bugs.

Following a bottom-up approach, there are at least four levels of testing, as shown in figure 4.13:

1. **Unit Testing**: is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed.
2. **Integration Testing**: individual units are combined and tested as a group. The purpose of this level of testing is to expose defects in the interfaces and in the interactions between integrated components or systems.
3. **System Testing**: at this level a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. **Acceptance Testing**: the last level, where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

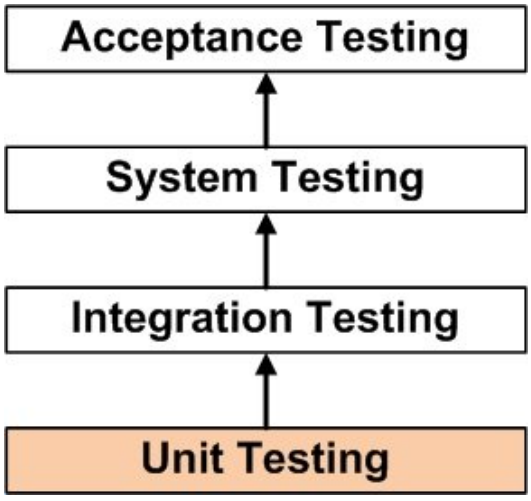

Figure 4.13: Testing methodologies.

What I did for the HTTPServer anomaly detector is to write a complete suite of Unit tests. This of course is a best practice when new software is developed from scratch, but it is also mandatory because, according to Lastline's coding policies, a piece of code cannot be used inside a product if it has not been properly tested and the overall code coverage is at least 90%. This is why, among all the testing levels, I will focus on the first one.

In Unit testing, a unit is the smallest testable part of any software. It usually has one or a few inputs and a single output. In object-oriented programming, the smallest unit is often a single method, but it can also be a whole class. By writing tests first for the smallest testable units, then the compound behaviors between those, one can build up comprehensive tests for complex applications. Performing using testing provides several advantages:

- Problems are found early in the development cycle. The cost of finding a bug before coding begins or when the code is first written is considerably lower than the cost of detecting, identifying, and fixing the bug when the code is in production. Bugs in released code may

lead to the crash of some components and, especially for critical applications such as health and banks, this can cause risks to the end user and a potential loss of money.

Moreover, the process of writing the test while developing the code to be tested itself forces the programmer to think about all the possible inputs and outputs and about the intended behavior, thus enhancing the overall quality of the product.

- It can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, the next level of software testing, integration testing, becomes much more straightforward.

- Code is more reusable. In order to make unit testing possible, a piece of code needs to be self-contained. This means that a snippet of code becomes an independent module and it is much easier to reuse.

- Changing and maintaining code becomes easier. If good unit tests are written and if they are run every time any code is changed, it is possible to immediately identify any bug introduced due to the change. As a consequence, also debugging is easier because, when a test fails, only the latest changes need to be investigated. Instead, with testing at higher levels, changes made over a consistent amount of time (days or even weeks or months) need to be analyzed.

On the other hand, there are also some drawbacks. For instance, unit testing can only show the presence or absence of some errors, but they cannot prove a complete absence of errors. This because it is impossible to evaluate any execution path for a program, especially if the code base is very large (as it happens for every consolidate industrial product). This is the reason why unit testing needs to be combined with the other layers described in figure 4.13. Moreover, another problem one faces when writing the tests is the difficulty of setting up a realistic and useful environment. It is necessary to create relevant initial conditions so the part of the application being tested behaves like it would in a real world scenario. If these initial conditions are not set correctly, the test will not be realistic and it would be useless.

When writing a unit test, it is important that each test case is run independently, in order to isolate issues that may arise. It is not as easy to create unit tests when the unit interacts with something external to the module itself. For instance, if a class depends on a database, the test should not query directly the database but, instead, it should adopt an abstract interface around that database connection and use some kind of replacement for the database. Substitutes such as method stubs and mock can be used to assist testing a module in isolation.

Following the company policy for testing the code, I used mock objects to achieve module isolation. Mock objects have gained a lot of popularity in the last years and they are one form of special case test object, one that enables a different style of testing. In fact, they are often confused with the more common concept of stub, even if there exists two main differences. On the one hand there is a difference in how test results are verified: a distinction between state verification and behavior verification. On the other hand it is a whole different philosophy to the way testing and design play together [11].

Commonly, the process of writing a test includes four phases:

1. **Setup**: the environment for a realistic execution is created.

2. **Exercise**: the test is actually performed, meaning that all the operation required by the test are executed on the unit.

3. **Verify**: confirm that the expected conditions are satisfied, by checking if the exercised method carried out its task correctly.

4. **Teardown**: perform all the actions needed to restore the state of the system as it was prior the execution of the test, such as freeing the memory from the used objects. Sometimes this phase can be implicit, a task left to the garbage collector in some languages (like python or java).

The target of a unit test is often a single object, which is named differently in literature like object-under-test or system-under-test. In the following it will be used the widely accepted term system-under-test, abbreviated as *SUT*, presented by G. Meszaros in [5]. A SUT usually needs to interact with one or more objects that are outside the scope of the test itself and these are called *collaborators*.

As introduced above one style of testing uses **state verification**: it determines whether the exercised method worked correctly by examining the state of the SUT and its collaborators after the method was exercised. This approach is generally chosen when using stubs.

On the other hand mock objects enable a different idea to verification, **behavior verification**, which checks if the SUT object operates as expected. This method is slightly different from the previous one. To begin with the Setup phase it is divided in two: data and expectations. The data part sets up the objects we are interested in working with, like it happens for state verification. However, the objects that get created are different. In fact, the SUT is the same, but the collaborators are mock objects (an instance of the Mock class), not the objects used in release. The second part of the setup creates expectations on the mock object. The expectations indicate which methods should be called on the mocks when the SUT is exercised [11].

Secondly, once the setup phase is done and the expectations are in place the exercise phase gets executed and it is the same as the state verification. What differs, instead, is the verification phase, which is divided in two like the setup phase. In the first step the conditions required by the test are verified as above. In addition, also the mock objects are verified checking that they were called according to their expectations.

The key difference then is how the verification is done. With behaviour verification the check is done by telling the mock what to expect during setup and asking the mock to verify itself during verification.

The vocabulary used to refer to these objects that replace the real ones to perform tests is very wide and often misleading in literature. Meszaros in [5] uses the term *Test Double* as the generic term for any kind of pretend object used in place of a real object for testing purposes. Moreover he introduces five particular kinds of double:

- **Dummy** objects: objects never actually used and often needed just to fill the parameters in a method.
- **Fake** objects: objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (for instance an in memory database example) [11].
- **Stubs**: crafted to provide custom answers when a call is made during a test and they usually do not respond to any call outside of the scope of the test.
- **Spies**: similar to stubs, they also record some information based on how they were called.
- **Mocks**: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Among all these kinds of double, only mocks make use of behavior verification. A stub usually focuses on state verification, even if it can go for behavior verification if some particular expedients are applied.

Chapter 5

Results

Evaluating the performance of a system of such complexity is not an easy task. In order to provide quantitative measures as exhaustive as possible, I tested my anomaly detector using different techniques and in multiple scenarios, so that the obtained results are meaningful. Regarding the comparison with the work presented in literature, on one hand it is possible to compare the performance with the product developed by Kruegel et al. in [1] but only partially, because the data sets they used for their analysis are not available. On the other hand, the system presented by Krueger et al. in [2] has a different structure and different purposes with respect to my work and the one in [1], because it mainly aims at healing the malicious inputs instead of reporting intrusions. As a consequence, the scope of this product is different and a comparison would not be very significant.

To begin with, I divided the performance tests in two main categories: *offline* and *online* tests. The first class deals with data collected from different sources and stored on disk, ready to be processed at any time. On the other hand, instead, online tests process the traffic as soon as it is received and nothing gets stored on stable storage. More details are given in the related sections.

Before running the tests, the system is configured according to the following combination of parameters (that has proven to be the most effective):

- HTTPServerPlugin triggers the flush every 1000 processed web request.
- Each AttributeModel requires a minimum of 1 client with one or more fitted sub-classifiers in order to start returning predictions.
- Each LengthClassifier requires at least 10 samples to be ready to fit.
- Each DistributionClassifier requires at least 10 samples to be ready to fit.
- Each DataTypeClassifier requires at least 15 samples to be ready to fit.

5.1 Offline Tests

Offline tests are a good indicator of the performance of a system because they allow to handle data sets in a much easier way than online tests, because they are not volatile but they are saved in one or more files. Therefore, the results are reproducible because the data do not change overtime and multiple measurement techniques can be used to assess the behavior of the system using always the same sample of inputs but from different points of view. As a consequence, I focused mainly on these kind of tests to collect the results of my anomaly detector.

More in details, I collected the data needed for the tests following two approaches. On hand I developed a web crawler in order to quickly generate HTTPS traffic (notice HTTPS and not HTTP) towards web server of my choice; on the other hand I directly downloaded real customer data sniffed by Lastline sensors and I used them to validate my work on real world traffic.

One thing to notice is that here the analysis performed by my anomaly detector is slightly different from what happens when it processes real customer data. In fact, when performing online tests, each web request is processed as soon it is received. Depending on the traffic load of the customer, this means that between one web request and the following one there may be also a consistent interval of time. Since the whole system (cleaning process, plugin flush, detector scheduling) is somewhat coordinated by time constraints this can lead to different behaviors with respect to offline tests.

In the offline setup, in fact, the anomaly detector processes the web request as fast as possible (because all the data are already available), the flushing process does no longer depends on the time but it is triggered periodically after a certain amount of web requests are analysed and the detector is scheduled only once, after the plugin has handled the whole data set. This differences can skew the results and this is why also online tests are performed.

5.1.1 Web Crawler

I wrote a simple web crawler using the well-known python framework *scrapy* [10]. A Web crawler, sometimes called a spider or spiderbot and often shortened to crawler, is a bot (i.e. an automated process) that systematically browses the Internet according to a given policy.

The script I wrote takes as input a domain name and an URL. Then, starting from these two seeds, it parses the initial web page and recursively visits all the URLs contained in it if they belongs to the specified domain. I ran the crawler using the Twitter domain (*twitter.com*) and a Twitter page as starting seeds and I collected roughly an hour of traffic.

Notice that the traffic generated towards the Twitter web server is HTTPS and, therefore, encrypted. In order to have access to the parameters and values of GET requests I need first to decrypt the traffic and analyze it in clear text. Of course, normally this is not possible. To face this issue, I downloaded and installed a Lastline sensor on my working laptop and I configured it to act as an explicit proxy, as detailed in section 2.3.2. More in details, working in this operation mode, the sensor is able to perform TLS decapsulation on the HTTPS traffic (i.e. it is able to decrypt the proxied traffic and process the content in clear text).

Therefore, when the sensor receives a web request from the crawler, it decrypts it and uploads the content to the corresponding Lastline Manager, that will store the request into a database. Afterwards, I was able to access this database, download all the data (in JSON format) and provide them as input to the HTTPServer anomaly detector.

Notice that this approach presents another limitation (on top of the ones described above): since the traffic is generated by one machine alone, there will be one client only in the data set and, as a consequence, the voting system implemented in the AttributeModel class is not used in practice.

The result of the analysis of 11718 web requests are provided in figure 5.1. The bar chart is read in this way: over 8820 parameters value analyzed by the HTTPServer anomaly detector in testing mode (i.e. after having considered done the training and validation phases), 20 values were considered anomalous and flagged as false positives. All the detections are considered false positives because the crawler simply recursively follows the links starting from a seed web page. Therefore, any web request is legitimate, unless the web page has been infected with malicious content (but this is unlikely for the test cases I used). As a consequence, this traffic can be used only to assess the number of false positives generated by my anomaly detector.

From the chart it is possible to quickly compute the false positive rate (FPR), defined as:

$$FPR = \frac{FP}{FP + TN} \tag{5.1}$$

where FP is the number of false positives and TN is the number of true negatives. For this data set:

$$FPR = \frac{20}{8820} = 0.0023 = 0.23\% \tag{5.2}$$

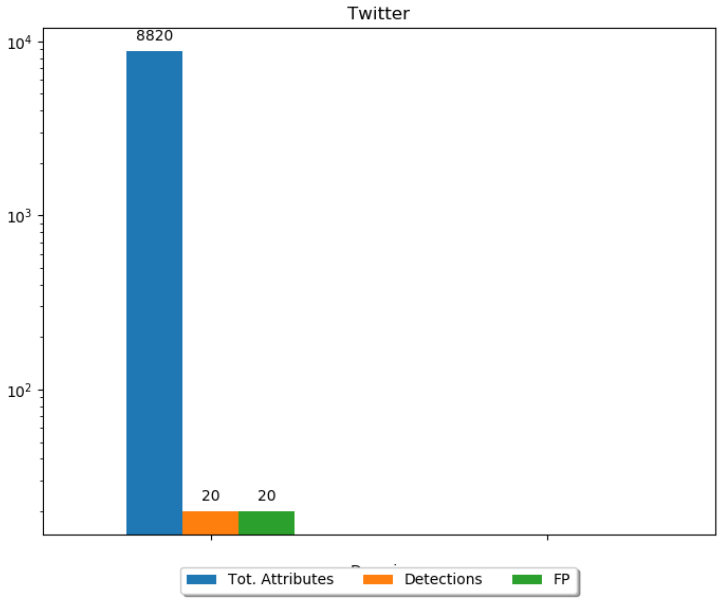

Figure 5.1: Results from the Twitter domain.

In comparison with the FPRs obtained in [1], with values between 0.01% and 0.07%, this result seems much worse but remember that the system worked with limited functionalities with a data set that does not really represent the real world traffic that targets a web server.

However, this was the first experiment I performed when I was not able to collect actual customer data because I did not have the permissions to access the database during the first months of my internship. Once I got these permissions, I was able to better evaluate my anomaly detector and to simulate more complex and complete test environments.

5.1.2 Offline customer data sets

Lastline provides two operation modes for its products: *hosted* and *on-premise*. When configured as hosted, all the data collected by the system components (mainly coming from the sensors) are uploaded to a Lastline Manager which is maintained by the company itself and the data are stored into a database *hosted* in the cloud. This means that Lastline has visibility over all the traffic entering and exiting the networks of the customers with an hosted license.

On the other hand, certain customer deal with very sensitive information (think about banks or military companies). Therefore, they do not want or they cannot share these data with Lastline and, to cope with this issue, they purchase an on-premise license. In this configuration, all the traffic is handled by a Lastline Manager that belongs to the customer itself and also the database is local and it resides in the customer network. In this way, Lastline is able to provide all its services without having any insight on the traffic, thus avoiding all the possible controversies, for instance those privacy related.

Once I got enough permissions, I was able to access the database that stores the traffic targeting the network of the customers with an hosted license. Among all the customers, I chose two web servers for my analysis: one with a public IP address and one with a private one.

The first address belongs to a web server inside the network of the Tokyo Institute of Technology. The choice was driven by the fact that the university exposes to the Internet an huge network (with a /16 netmask). This means that the web server is reachable by any host in the Internet and, as a

consequence, it is subject to receive any kind of traffic, both malicious and benign. For instance, I noticed that this web server in particular, is often target of vulnerability scanners, an optimal source of malicious inputs.

The second data set, instead, correspond to a JetBrain "Floating License Server", an on-premise application that you can install in your company's infrastructure to enable automatic distribution of JetBrains floating licenses [12]. Being a web server with a private IP address it cannot be directly contacted from the public Internet and, therefore, it receives an amount of traffic much smaller than the one towards the first web server. Moreover, it is less likely to receive malicious inputs because the server can be reached only from within the customer network, meaning that if it receives suspicious data, they probably come from an infected internal host or they are the result of misconfiguration errors.

For each web server, I downloaded two separate data sets containing traffic equivalent to about three working days (I avoided the weekends because, as expected, the activity is much less). Then, for each of the data sets, I applied a *cross-validation* algorithm to ensure that the system has got most of the patterns from the data correct, and it is not picking up too much on the noise, or in other words it is low on bias and variance.

Cross-validation helps dealing with problems such as underfitting (the system is not able to extract enough features from the training data and performs poorly) and overfitting (the system handles perfectly the training set but it is not able to generalize to different data). There are several different cross-validation techniques, and I used the well known *KFold*.

KFold splits the input data set into k smaller chunks and then it uses k-1 chunks for training and one for validation. The general procedure is the following:

1. Shuffle the data set randomly.
2. Split the data set into k groups.
3. For each unique group:
   (a) Take the group as a hold out or test data set.
   (b) Take the remaining groups as a training data set.
   (c) Fit a model on the training set and evaluate it on the test set.
   (d) Retain the evaluation score and discard the model.
4. Compare the metrics derived from each iteration.

Notice that each sample is given the opportunity to be used in the hold out set 1 time and used to train the model k-1 times. I set k to 10 because this is the value that has been found through experimentation to generally result in a model performance with low bias and modest variance.

However, due to the way my anomaly detector is designed, the concept of cross-validation does not apply properly. In fact, given a data set, the goal of a cross-validation technique is to split that data set into a training set and a validation set and to measure the performance of the model. Since HTTPServer does not have a clear separation between training and validation (the sub-classifiers continue training until they are not ready to make a prediction and the AttributeModel is always in training mode), cross-validation can be used just to perform a permutation of the inputs given to the model and to see if the false positive rate significantly increases or decreases.

Figures 5.2 and 5.3 shows the results obtained by providing as input to HTTPServer the two data set containing the traffic towards the Tokyo Institute of Technology. Each data set contains 1000000 HTTP GET requests with a non empty query string, meaning that all the web requests are used by the system.

In the chart there are ten group of three bars, one for each round of the KFold algorithm and, for each round, I collect the number of attribute values for which the system gave a prediction (i.e. after having completed the training phase), the number of suspicious values and the number of false positives (that I computed manually after each iteration of the algorithm).

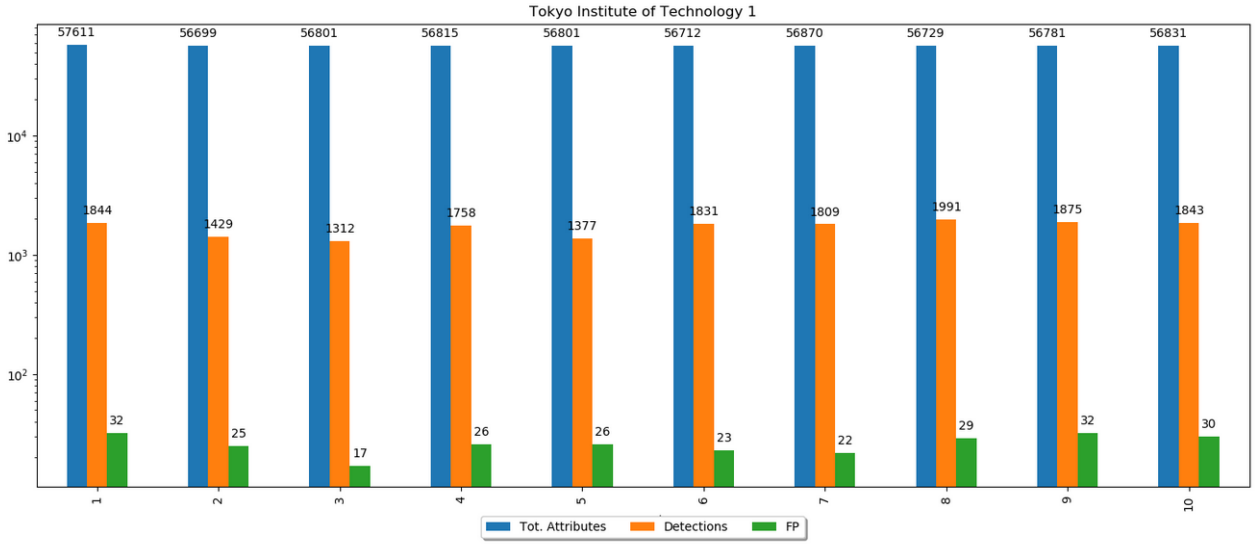

Figure 5.2: Tokyo Institute of Technology 1

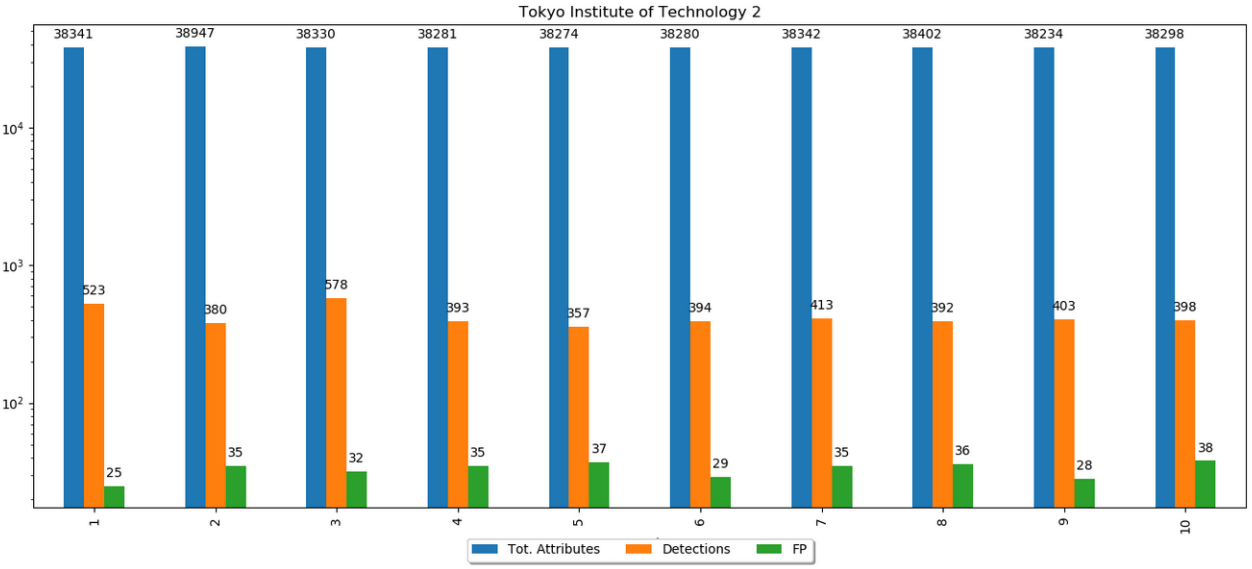

Figure 5.3: Tokyo Institute of Technology 2

At a first glance, it is possible to observe that, for both the data sets, the trend of the metrics is practically constant. From a quantitative point of view, the minimum, maximum and average false positive rates for both the data sets are provided in equations 5.3 and 5.4 respectively.

$$min\_FPR\_1 = 0.03\% \qquad max\_FPR\_1 = 0.06\% \qquad avg\_FPR\_1 = 0.05\% \tag{5.3}$$

$$min\_FPR\_2 = 0.07\% \qquad max\_FPR\_2 = 0.09\% \qquad avg\_FPR\_2 = 0.08\% \tag{5.4}$$

I applied the same logic to the other two data sets, each of them containing 10000 records collected in the same amount of time as the previous example. Notice the number of web requests (two order of magnitude less than the data set from the Tokyo Institute of Technology) that is in line with the fact that this second web server is private and it receives much less traffic.

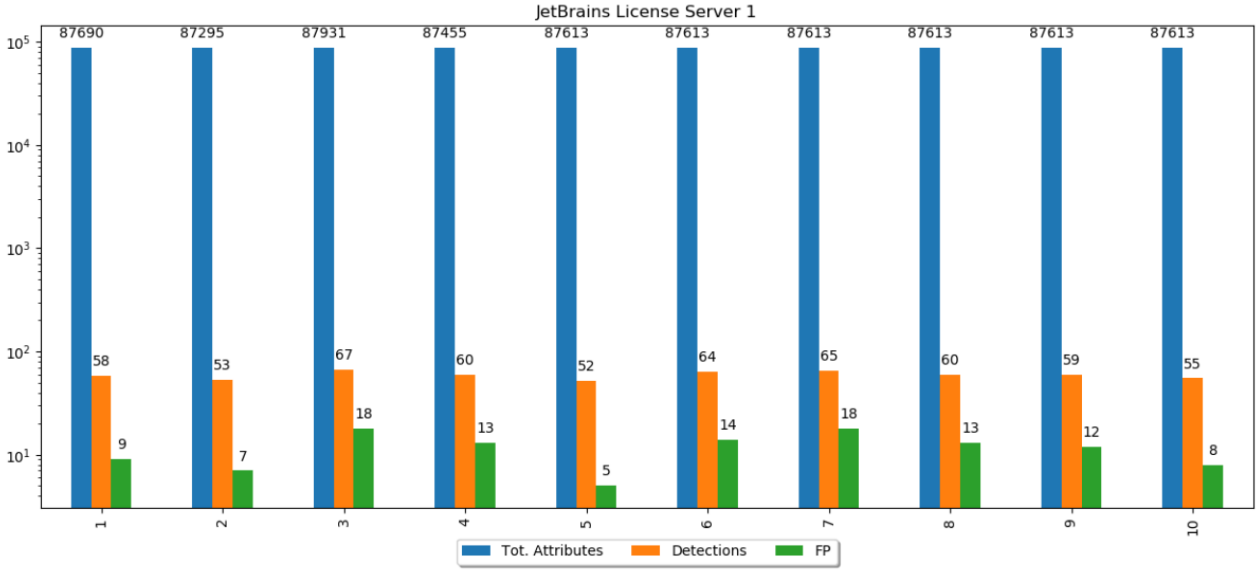

Figure 5.4: JetBrains License Server 1

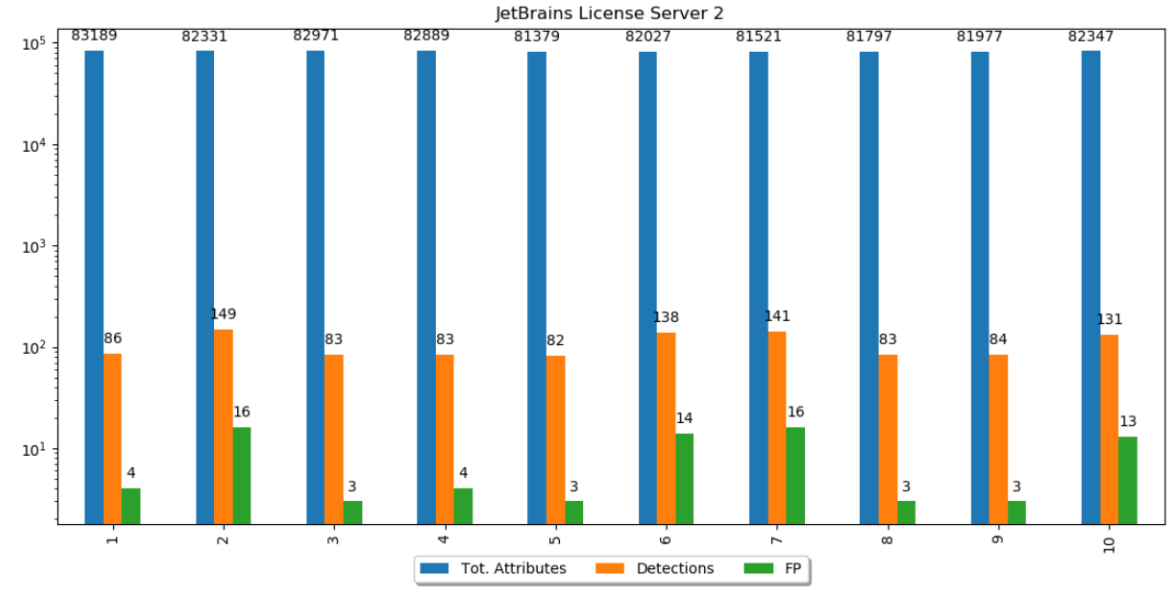

Figure 5.5: JetBrains License Server 2

I obtained the results reported in figures 5.4 and 5.5 respectively.

Like for the first example, the quantitative measures of the false positive rates are illustrated in equations 5.5 and 5.6.

$$min\_FPR\_1 = 0.006\% \qquad max\_FPR\_1 = 0.02\% \qquad avg\_FPR\_1 = 0.01\% \tag{5.5}$$

$$min\_FPR\_2 = 0.004\% \qquad max\_FPR\_2 = 0.02\% \qquad avg\_FPR\_2 = 0.01\% \tag{5.6}$$

As expected these latter results look better than the previous ones, always because this second web server should be less exposed to potential attacks. Moreover notice that, of course, the pattern

of the traffic targeting the two sample web servers is very different. In fact, when processing the data set from the Tokyo Institute of Technology, even if the number of web requests is much higher, the number of attribute values for which a prediction is given is in the order of 50 thousands for the first batch and 30 thousands for the second.

The internal web server, instead, on average receives web request with a bigger number of parameters and therefore, even with only 10000 web request in each data set, the number of predictions is in the order of 80 thousands for both samples. This means that it is quite hard to compare the performance of the system with data sets related to different sources. because the traffic pattern is very dependent on the type of the applications exposed by the specific web server and on how it interacts with its clients. However, if we want to make an analogy with the results obtained by Kruegel et al. in [1], the performance of the HTTPServer anomaly detector are similar and, in some scenarios, even better.

5.2 Online Tests

Every LLAnta module is designed to process the input as soon as they are received and, usually, it means as soon as the traffic is sniffed by the sensor. The working environment is, therefore, very dynamic.

Once the HTTPServer anomaly detector was ready to go in production, it was at first deployed in a testing environment, where a bunch of pcap files are replayed in loop to generate some traffic. The goal of this first stage is to make sure that the detector works, in the sense that it does not crash and it does not generate too many alerts, symptom that something is off. Yet at this early phase there may be some issues to face that are not highlighted by the unit tests. For instance, I needed to solve a couple of problems, such as a wrong use of the configuration file (that was causing a big waste of space on the disk) and a bug in the serialization (that was crashing the python module in charge of storing the context on disk and, as a consequence, nothing was saved on stable storage).

After these minutiae are fixed and the detector is stable for a couple of days, it is deployed also in one host where a LLAnta instance is running. This host controls the networks of few real customer and receives the traffic from the installed sensors. The system is then monitored for roughly a week and, if everything looks fine and the module works without creating problems to the whole LLAnta ecosystem, it is finally deployed in all the LLAnta instances processing the inputs from all the Lastline customers.

During all the stages, checking the behavior of the system is not an easy task. For every host where there is a running LLAnta instance I can access a dashboard collecting some metrics, such as:

- Plugin processing time.
- Flush duration.
- Detector processing time.
- Number of alerts generated by the detector.

However, all these statistics are related to the LLAnta instance active on the analyzed host. This means that, since the host receives traffic coming from more than one customer, the information of different customers is mixed together. For instance, regarding the number of generated alerts, it is impossible to tell which alert was generated by which customer by just looking at the dashboard and without going to manually check in the database. Notice that, since each alert and the related information appears in the customer user interface, it would be much easier to access the customer dashboard and investigate from there but, since my module is new, it is not currently supported in the UI.

Therefore, the only thing I could do was to monitor the behavior of the system by looking at the dashboard and to make sure that the various graphs made sense. Given the huge amount of

processed traffic, it is impossible to manually investigate each event to assess if it is a false positive or not.

Figure 5.6 shows a graph with the alerts generated by the 11 customers handled by a LLAnta instance. Over 30 days of analyzed traffic there was a maximum of 24 alerts and an average of 4 alerts per day. Notice that these number are perfectly manageable even if all the alerts were actually false positives, because a network analyst can quickly investigate them without loosing hours of work.

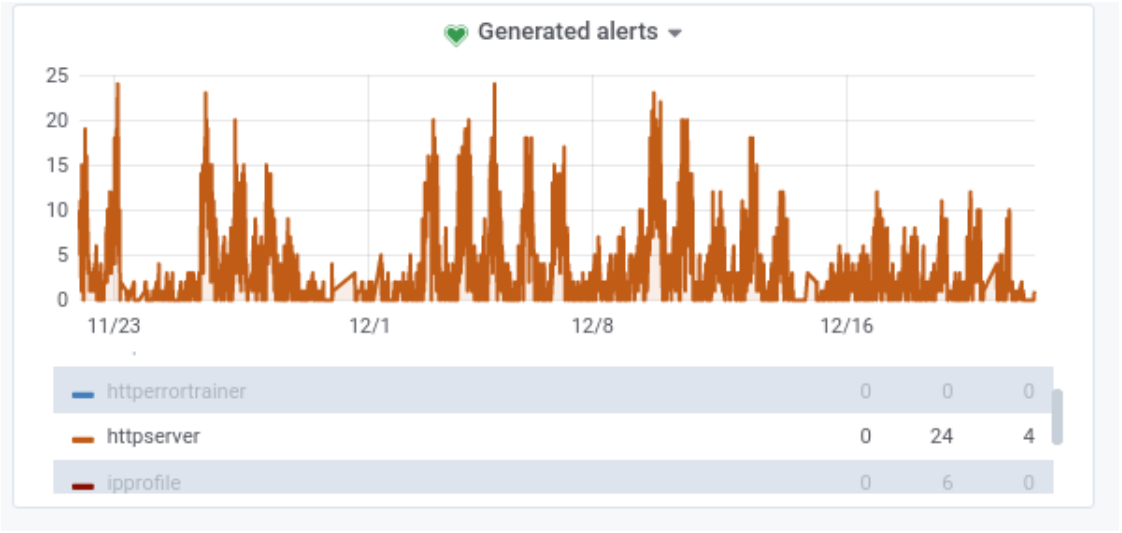

Figure 5.6: Alerts generated over 30 days.

Chapter 6

Conclusions

During my internship at Lastline I developed HTTPServer, a modular anomaly detector system for HTTP traffic that has been successfully integrated in the company's network analysis tool, LLAnta. For this job, I took inspiration mainly from the ideas presented by Kruegel et al. in [1] and by Krueger et al. in [2].

However, unlike the other systems in literature, HTTPServer is specifically designed to process traffic in real time as fast as possible, in order to be able to report potential breaches as soon as they are detected. For these purposes, it is essential that the single components (HTTPServerPlugin, HTTPServerContext and HTTPServerDetector) are light enough to keep up with the traffic load targeting the customer network. The more time HTTPServer requires for its processing, the more delay is introduced in LLAnta, which can lead to an always increasing degradation of the performance. This can eventually cause the drop of potential relevant data if it cannot handle the speed at which the data come, that is a severe flaw for a cybersecurity company.

Moreover, also the context in which my anomaly detector is designed to operate is different with respect to the literature. In fact, both the ideas in [1] and [2] propose their solutions to a supervised machine learning problem. This means that there is either the possibility to access a clean training set (i.e. composed by benign data only) as in [1], either the system goes through a setup procedure before start working properly where a system administrator can properly tune all the parameters and deal with false positives, as in [2]. In both cases, there is an a priori knowledge of the type of traffic is going to be used to train the model.

HTTPServer, instead, performs unsupervised machine learning. The traffic targeting a certain web server is specific to that customer and to the type of the exposed application. As a consequence, besides truly malicious inputs, what is considered anomalous for one customer can be considered perfectly benign for another one. For instance, a certain web server could expect to receive an attribute "id" as a numerical value only, while another one could treat the same parameter as an alphanumerical value.

In this working scenario it is impossible to have knowledge of the kind of traffic is going to be processed by the anomaly detector and, especially for web server exposed to the Internet, it is not unlikely that malicious traffic is received during the training phase. By not being able to assure that the training phase only deals with clean data, it is a quite complex challenge to allow the detector to recognize benign inputs from malicious ones. HTTPServer is able to adapt to the traffic of the particular web server it is protecting and it does not require any external data set to work properly. Thanks to its design and the introduction of the client voting system for an AttributeModel, it is very robust to the presence of outliers and the obtained results are at least as good as the ones presented in [1], even in a more difficult working context. This innovative feature is the main contribution I achieved during my internship.

The whole system should work under the assumption that the majority of the clients contacting the web server produces benign traffic, so a potential malicious client will not affect the creation of the model in a decisive manner. Moreover, I expect malicious inputs to have different characteristics and to follow different patterns than benign ones. In particular, these where my hypothesis

regarding each single sub-classifier that profiles the traffic:

- **LengthClassifier**: benign attribute values should have a length that does not vary too much between requests associated to a certain resource path. Attacks such as buffer overflows or Cross-Site scripting require to deliver a large payload that often exceeds the length of legitimate parameters.
- **DistributionClassifier**: parameter values usually show a regular structure, for instance by containing only printable characters. However, it is common in BOF attacks to send binary data with a completely different distribution. This is true also for directory traversal exploits, where there is a repetition of the dot character, which can be unusual, and for SQL injection attempts.
- **DataTypeClassifier**: this classifier determines the data type of a certain parameter. If an attribute belongs to a particular class it is very unlikely that values not matching the data type are benign, even if they do not show attack-specific patterns (such as binary or large payload). This can be useful to spot attackers probing a web server for vulnerabilities by sending random inputs.

If all the above assumptions hold, and in general they do since they are quite reasonable, HTTPServer should be able to detect not only truly malicious traffic but also all the inputs that show a different pattern from the established benign one. This can include, for instance, client sending anomalous payloads due to misconfiguration errors.

Naturally, the system is not perfect, it presents some weaknesses and there is a lot of work that can be done to improve it and enhance the detection performance. One of the main concerns, for instance, is finding the optimal combination of the configuration parameters. It is hard to set the thresholds once for all, because Lastline has customers all over the world and the traffic generated in Japan is completely different from the one generated in the US (think about the characters encoding for instance).

As mentioned before, also the traffic changes according to the type of customer. Big companies such as banks receive an huge quantity of web requests from a lot of different clients. In this case the voting system should work good and it is fine to set up an high threshold on the number of samples for the training of the classifiers. On the other hand, some customers also want to protect internal web servers, that are not exposed to the public Internet, hence they receive much less requests from a limited number of clients. In such situations it should be better to set lower thresholds, otherwise it could take a very long time for the classifiers to start returning predictions.

There are multiple ways to achieve this goal:

- Before deployment is it possible to download some traffic belonging to the specific customer and run HTTPServer offline (as in the offline tests in section 5.1.2). The customer can then receive the results of the analysis and assess the system performance.
- After deployment, the customer itself can keep monitoring the behavior of the system and, after a certain amount of time, a network analyst, by knowing the traffic that flows in its own network, can decide if tuning the configuration setting is needed or not. A tune of the parameter is generally recommended every once in a while, as the traffic targeting a certain web server can vary over time.

Moreover, like many other companies, Lastline offers a trial of the products to potential customers. The trial lasts about a month and during this period the customer can verify if the system actually works or not and if it is satisfied by the overall provided protection.

Another aspect to consider is the technique used by the DistributionClassifier. I noticed that sometimes the predictions returned by this classifier are unreliable if compared with the other two classifiers. As a consequence, I have always set an higher number of minimum training samples for it in all my test. This, however, can only be a partial fix and it would be nice to implement

an automatic mechanism that is able to decide during the validation of a classifier if it is actually ready to start operating in testing mode or if it is better to keep training.

There are several improvements and extensions I plan to do over the next months, such as:

- Improve logging. Log files are the first and easiest way to have an initial understanding of what is going on (without going to check the detector state on disk, which can be very large, especially after a week or more). My anomaly detector already provides some basic logging but this is clearly not enough and, now that I have an insight on real traffic, I want to make the records as meaningful as possible.
- Based on the logs and on the analysis of the serialized state, I would like to tune better the threshold used by the detector. As described also in the previous reports, this is a very challenging task because the network traffic targeting different customers varies widely and it is almost impossible to find a "golden rule" that works in every scenario.
- Expand the HTTPServer anomaly detector including also the HTTP headers in the processing.
- Improve the analysis by adding more classifiers to the system (beside the LengthClassifier, the DistributionClassifier and the DataTypeClassifier). Thanks to the modular design of the system, the inclusion of new sub-classifiers requires minimal changes (only to the AttributeModel class).
- Allow the detector to wipe an AttributeModel from the HTTPServerContext if it detects that the predictions are wrong (too many false positives).
- Add support for the POST HTTP request.

This latter idea is actually already in progress but, at the moment of writing, still not complete because it involves changes to many components.

As introduced in section 2.3, Lastline develops proprietary Suricata patches in order to extend the information sniffed from the network traffic. Therefore, the first part of this task was about analyzing the related Suricata code, understanding it as much as possible (given the huge code base) and finding a way to add the body of the HTTP POST requests in the data generated by suricata-eve.

Once the sensor is able to extract the data I need and send them to the Lastline Manager, I still have to extend the Manager to accept the new information and modify my LLAnta module (in particular HTTPServerPlugin) to handle it. All these jobs represent a very good exercise that allowed me to see and work on the whole Lastline pipeline.

I successfully added a patch for Suricata that extends the module dedicated to log information (in JSON format) when an alert or an event for the HTTP protocol is generated. The new feature can be enabled by setting to *true* the corresponding *dump-req-body* option in the configuration file, *suricata.yaml*.

When an HTTP event or an alert is generated, if it is a POST request and the content-type header value is *application/x-www-form-urlencoded*, the patch adds to the log also the request body, encoded in Base64. The request body is encoded to avoid issues when transmitting binary data over the network (for instance some bytes may be interpreted as control characters). Moreover, The size of the body is limited to 1MB in order to avoid uploading too much information.

The choice of the constraint on the content-type header value is motivated by the fact that we want to upload only the parameters sent to a web server, like it happens for GET requests. For instance, we want to avoid uploading a file (usually represented by the content-type *application/octet-stream*), because other Lastline's components already take care of analyzing and securing this kind of input.

Once the the information is dumped, it is serialized into a Google Protocol Buffer (protobuf). It is then processed by suricata-eve, that decodes the input and sends it to the Lastline Manager.

This will process the data and, if appropriate, it will dispatch them to the corresponding LLAnta instance and to my detector. I still need to implement this second part and make HTTPServer able to handle this new data.

One final consideration is related to the work done by Krueger et al. in [2]. When reading the paper, I found really interesting and innovative the concept of healing action. I was tempted to implement some of those techniques also in HTTPServer but I did not because the purpose of my system is to work as an IDS (hence just to report the alerts without actually performing any action).

Bibliography

[1] C. Kruegel, G. Vigna, W. Robertson, "A multi model-approach to the detection of web-based attacks", Computer Networks, Vol. 48, No. 1, August 2005, pp. 717-738, DOI 10.1016/j.comnet.2005.01.009

[2] T. Krueger, C. Gehl, K. Rieck, "TokDoc: A Self-Healing Web Application Firewall", SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing, March 2010, pp. 1846-1853, DOI 10.1145/1774088.1774480

[3] P. J. Rousseeuw "Tutorial To Robust Statistics", Journal Of Chemometrics, Vol. 5, No. 1, January 1991, pp. 1-20, DOI 10.1002/cem.1180050103

[4] B. Iglewicz and D. Hoaglin, "Outlier Labeling" in the book "How to Detect and Handle Outliers" edited by Edward Mykytka, ASQC Quality Press, 1993, pp. 9-13,

[5] G. Meszaros, "Principles Of Test Automation" in the book "xUnit Test Patterns: Refactoring Test Code" edited by Martin Fowler, Addison-Wesley, 2007, pp. 39-48,

[6] The State of Web Application Vulnerabilities in 2018, https://www.imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2018/

[7] Cost of a Data Breach in 2019 Report, https://securityintelligence.com/posts/whats-new-in-the-2019-cost-of-a-data-breach-report

[8] Managed Security Service Provider, https://www.gartner.com/it-glossary/mssp-managed-security-service-provider

[9] Universally unique identifier, https://en.wikipedia.org/wiki/Universally_unique_identifier

[10] Scrapy, https://scrapy.org/

[11] Mocks Aren't Stubs, https://martinfowler.com/articles/mocksArentStubs.html

[12] Floating License Server, https://martinfowler.com/articles/mocksArentStubs.html

[13] Lastline Security Challenges, https://www.lastline.com/use-cases/

[14] Modified z score, https://www.ibm.com/support/knowledgecenter/en/SSEP7J_11.1.0/com.ibm.swg.ba.cognos.ug_ca_dshb.doc/modified_z.html

[15] URLparse, https://docs.python.org/2/library/URLparse.html