Master of Science Thesis

HTTP Application-level Intrusion Detection and Prevention

Fernando Álvarez Cabrera

Supervisor: Professor Robin Sharp

I would like to give special thanks to my parents Miguel A. Álvarez and Pilar Cabrera Adriano. Thank-you for having supported me throughout all of these years.
I would also like to give special thanks to two dear friends of mine; David Delgado Gómez for all his help and support during my stay in Denmark and to Paul Peter Schuster, "Just keep swimming!".

Abstract

Within computer security, intrusion detection is one of its key players. Intrusion detection is commonly carried out at the lower levels of a network's architecture. For example, the inspection of a TCP/IP packet's properties. Intrusion detection systems have tried to analyze content, for some time now, at an application layer of the network's architecture. The results of application-level analysis have not had much success. This document presents an application-level intrusion detection system. The application-level protocol subject to analysis is HTTP. The system is based on neural network technology for categorizing classes of known attacks. The system is stateful enabled i.e. it is capable of correlating a sequence of suspicious HTTP requests with their HTTP responses in order to detect temporal patterns of behavior. The system also presents close to real-time analysis during the service of a client's HTTP request, making it a fast and robust preemptive analysis tool.

Acknowledgements

I would like to acknowledge my tutor, professor Robin Sharp. His patience and guidance helped a great deal throughout the development of this project.

I would also like to acknowledge the open-source community. Most of the software tools, if not all, used herein are within the Public domain. I sincerely hope that some day, I will be qualified enough to provide the community with valuable software. For someone else, like me now, to benefit from the availability and functionality of open-source software.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The problem that this M.Sc. Thesis will address, are the issues presented while analyzing application-level network traffic for intrusion detection. The application-level protocol that is the subject of analysis is HTTP. An application-level IDS (Intrusion Detection System) is presented. The IDS was developed, in parallel with this document, in order to prove the stated concepts. The system is based on neural-network technology, and is composed of a series of sub-systems intercommunicating to provide fast and precise intrusion detection. The analysis, design and testing of the individual sub-systems along with their inter-workings as a whole system, will be described in the following chapters.

1.2 Motivation

As information systems evolve, and the direct or indirect interaction of humans with these information systems increases, a necessity for secure communication mediums raises. Commonly the term secure communication, is only focused towards a user perspective i.e. the client-side within a client-server application context. Although, the integrity of data received by a particular process offering a service i.e. server-side, is just as important to the service's availability. How fault-tolerant are information systems towards erroneous data sent to them? Accounting that it could be a user sending erroneous information; due to a faulty application or with a deliberate intention in mind. Can agents be automatized to efficiently monitor the information flow between two or more parties and prevent the disclosure of restricted information?

Considering that modern network architectures such as the Internet, follow layered architectures e.g. TCP/IP figure 1.1. Information exchange between two or more entities happens at different conceptual levels. Ranging from raw physical data, to higher abstract application data, e.g. e-mails and HTTP requests.

There are a few techniques used to automatize an agent to monitor network traffic on such a network architecture. It may sound reasonable to have one agent or various sub-agents monitoring the different content from each layer. In the latter case, each sub-agent monitoring at the corresponding layer that they are built for, e.g. a network-level sub-agent monitoring raw

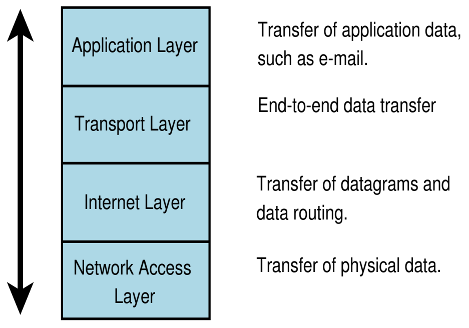

Application Layer
Transport Layer
Internet Layer
Network Access Layer
Transfer of application data, such as e-mail.
End-to-end data transfer
Transfer of datagrams and data routing.
Transfer of physical data.

Figure 1.1: Layered architecture of a TCP/IP network.

TCP, UDP and ICMP packages and an application-level sub-agent monitoring FTP transactions. Ideally, each of these sub-agents should not only monitor data flow at their level, but also intercommunicate. The data exchange between the sub-agents grants a more precise view of a network's state. An example of such an interconnected scheme is depicted in figure 1.2.

These monitoring agents are formally known as IDSs. Within a network schema, a network-level IDS is known as NIDS (Network Intrusion Detection System). These are designed to monitor transport layer datagrams and are, in many cases, unaware of data-exchange at higher levels of the network's architecture. The unawareness is due to NIDSs' requirements specification.

Within the NIDS family, an attempt has been done to incorporate different layer-level monitoring features. This multi-layered proposal raises an interesting issue within the field of intrusion detection. Are NIDSs capable of recognizing intrusions at a higher level of a network's layered architecture? For example, one aimed at taking advantage of a service's poorly designed protocol. Can the NIDSs understand the semantical properties, at an application-level of the network's architecture, of data-flow and not only its syntactical properties?

Up until now the most common method of analyzing application-level information, by NIDSs, is a regular expression (character sequence) search within network traffic. This method is also known as rule-based analysis or rule-based filtering. There are a few disadvantages with rule-based analysis; NIDSs will not categorize highly sophisticated intrusions at an application-level. They are unable to interpret the possible attack scenarios, for all available application-level services. This disadvantage leads to NIDSs presenting a high false-positive rate during analysis of application-level data i.e. categorizing abnormal but benevolent data-flows as suspicious ones. This high rate is due to application-level services widely varying in scope. Hence, data that for one type of service is of no particular interest may mean a security breach for another.

1.3 Structure

The structure of this M.Sc. Thesis is divided into eight chapters. Being chapter three through eight the ones that describe all aspects of the application-level IDS developed throughout this project. It is recommended that appendix A be consulted for clarifications on definitions and abbreviations while reading this document.

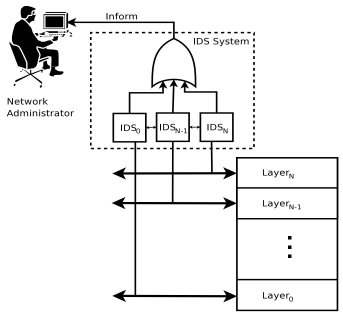


Figure 1.2: A set of IDS sub-systems interconnected providing precise views of the network's state.

Chapter two, provides some basic theoretical background used to introduce IDS notions and features. It starts off by providing some background on IDS paradigms. The CIDF framework is briefly described since it conforms to the main architecture that has been chosen for the application-level IDS. Then an introduction to application-data along with some examples, following with HTTP specific application-data connotations. This chapter provides the backbone for most ideas behind the system analysis carried out in chapter three. The IDS background provided should also correlate to a basic understanding of the problem domain.

Chapter three describes the analysis process including IDS requirements elicitation. In this section available tools under public and private domain are reviewed in order to get a broader idea during the requirements gathering phase. Some pros and cons are mentioned for each of the tools reviewed. In the last section of this chapter some neural network algorithms are mentioned. The IDS developed herein depends greatly on its classification capabilities, which is why a neural network approach is used. Then the algorithm used by the IDS will be briefly described.

Chapter four provides the system design describing the specific configuration chosen for the application-level IDS along with a system architecture. Not only one particular architecture is used for the IDS. The design is broken up into the various functional components and subsystems. All of these are described with different architecture types. Using different kinds of architectures, provided more flexibility during the design phase. All of the interconnections between system components are presented in this chapter. In the end, behavioral models for all functional components are either described with pseudo-code or depicted.

Some of the implementation issues that were encountered are presented in chapter five. Main class hierarchy diagrams are provided in this chapter. Why a certain style of programming was chosen will be described along with the associated programming concepts used. The system was developed using object-oriented techniques, and some examples of applied concepts are; how inheritance and polymorphism were integrated into the system to facilitate implementation tasks. The prototype implemented is not fully functional, it proves the stated concepts but is not integrated into a true networking environment. How the networking environment was simulated will be described in this section.

Chapter six presents, from the different tests performed, the most relevant ones that support all final conclusions. The physical environment i.e. what types of devices were used along with all auxiliary applications. How all testing data was obtained is also described e.g. how coherent neural network training and testing data was gathered to measure the accuracy of the IDS. The timings that the system achieves are presented in this chapter, e.g. the system's training time as well as the system's request testing time. In the last section of this chapter the system's detection efficiency achieved is presented.

Considering that the final system delivered is at a prototype stage, chapter seven presents some further developments that could be incorporated into the system. Some of these further developments would be an essential requirement if the system was aimed at guarding a set of heavily loaded Web-Servers. Some of these issues include the schema that could be used under a hand-off scenario. Hand-off scenarios are when multiple Web-Servers are used to service a particular site. Also some system kernel improvements are mentioned such as; on-line training and the fix for a DoS attack against the IDS itself.

The last chapter presents all final conclusions on several aspects of this project. For example, the feasibility of deploying the proposed system on a truly concurred web-server. After these aspects are discussed, a final statement is given on the general development of this project.

All additional software used or implemented for this project, is presented in the appendices. The user interface to these additional libraries i.e. how these libraries are used in user programs, is the main content of these last sections.

Chapter 2

Domain Description

2.1 Intrusion Detection

Intrusion detection is one of the main branches in computer security technologies. This discipline tries to effectively detect whenever users attempt or succeed, in accessing an unauthorized service provided by a particular system. The reasons for unauthorized accesses vary, among these, the most common is access to undisclosed information. IDSs should, in conjunction with other security components form part of a so-called secure network. IDSs along with cooperation from other security components, provide vital information to the network managers. Which helps in getting an overall picture of the activities happening in the network.

IDSs may be categorized by various properties, for example the type of analysis that the IDS performs, such as anomal or misuse (signature-based) analysis. Within misuse or anomal analysis, the activities that take place can be interpreted as a single events or a series of them. This leads to stateless and stateful analysis.

After reviewing the various components of the CIDF, IDSs may also be categorized by the types of countermeasures that are perform i.e. preemptive or passive actions. The localization of an IDS is also used as a valid classification property i.e. network-based or host-based IDSs. The last set of properties mentioned herein, is how an IDS behaves when it reaches a faulty state within itself. This means whether the system is fail-open or fail-close. All of the above mentioned including the CIDF model will be explained in the following sub-sections.

2.1.1 The CIDF Model

The CIDF (Common Intrusion Detection Framework) model is the essential set of blueprints under which most intrusion detection systems are built. It is composed of four main components which are depicted in figure 2.1.

Figure 2.1 also shows the interconnections among these components through which specific data flows. The different components represent an IDS's internal functional parts. The four components are:

- An event gatherer , which is constantly listening on an event medium.

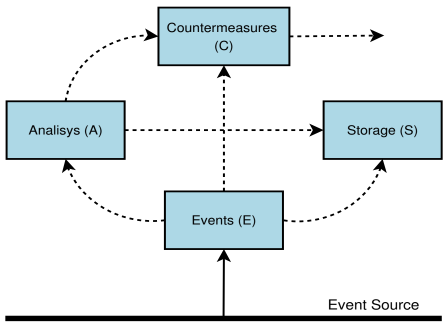


Figure 2.1: Main components in a CIDF Model.

- A storage component which logs all relevant IDS activity.
- An analysis component which decides if attention should be paid to a given event or not. This component will also trigger certain countermeasures depending on what it has concluded from an event.
- A countermeasure component, which will take measures against malicious events.

For trivial reasons, among all of these components, the analysis component is probably the most important. This component, given that the others work in accordance to their specifications, will determine the worth of an IDS. What is it really capable of detecting?

2.1.2 IDS Analysis Techniques

When an IDS is capable of stateful inspection, it takes into account the timing of events. Unlike stateless analysis, most rule-based systems, where if an event matches a given rule it is dealt with. The problem with stateless analysis is that highly sophisticated attacks are impossible to detect. Sophisticated attacks are conformed by a series of events not just one. So a chain of events must first be identified to successfully detect the intrusion. A good example of stateful attacks, are distributed DoS (Denial of Service) attacks. A problem that stateful IDSs present though, is that they can be the subject of the attack. A metaphor to help understand this problem, is the TCP SYN attack. Where a host will continue to send SYN TCP packages, even after receiving the corresponding ACK TCP package. In the attempt to overflow the destiny's SYN buffer.

A different type of analysis is when the IDS is capable of detecting usual (anomal) behavior in the system that it is monitoring. Anything out of these predefined bounds is detected, and the relevant countermeasures taken. The main problem with anomal analysis is that, it raises a high number of false-positives. Making it hard to distinguish between real attacks and benevolent data that is not common to the system. The counter-pose, is if the IDS is capable of detecting the individual attacks themselves (signature-based) instead. The problem with misuse analysis though, is that it is not possible to model all attack scenarios. With the growth rate that the Internet is subject to, new attacks are constantly arising. So the modelling of attack scenarios will always be a step behind, subject to what is already existent.

2.1.3 IDS Countermeasures

The type of countermeasures taken can differentiate an IDS by whether it be a preemptive or a passive system. When a system is only built to raise alarms and notify the appropriate identities, it is called a passive system. Passive systems are sensor-like systems, where if a determined event triggers a sensor an alarm is raised. In the event of an attack these systems generate data which is later used for forensics examination. Passive systems can be best seen as observers that are monitoring data between two or more parties. Most IDSs on the market today behave in a passive manner. One of the problems with these systems is that, it may take just one event to bring down the target of an attack.

Preemptive systems on the other hand, behave like filters. Where there is no time to raise alarms, the actions which cause faulty system behavior have to be spotted beforehand and contained before they reach their intended destiny. The main disadvantage that preemptive systems present is that in an excessive security policy, a service's availability can be affected, resulting in an annoyed group of users.

2.1.4 IDS Localization

The localization of an IDS is another important factor while categorizing. The main two categories are host-based systems and network-based systems.

Host-based IDSs, reside in the same machine where the service is being offered. These IDSs can have a very good idea of which data is being directed towards a given service. There are no network data desynchronization problems between the IDS and the destiny service with these IDSs. These IDSs are built into the application which they are monitoring, as a sub-system. This property allows the IDS to even monitor encrypted data being sent to the service. A depiction of the idea may be seen in figure 2.2. The only problem with these systems is that they cannot have a broader view of the events happening throughout the entire network. This affects distributed networks, where several machines are used to offer different services. Sophisticated intrusions sometimes take advantage of this network layout, and not only target single services but several of them. An issue that sometimes drives attention away from host-based IDSs, is the impact that they will have on the machine offering a given service. If it is a very concurred service, it may be necessary that it be offered by a single machine. Therefore, having another process monitoring on the same machine, may affect its performance.

The other type of IDS localization is network-based, also depicted in figure 2.2. These systems are laid out somewhere in the network, monitoring all traffic. Sometimes these systems are actual specific hardware devices that may only receive data through their network interfaces. This helps in making them invisible to the rest of the network, since no response will be obtained from them. The main problem that these systems present, which is also the main argument for the application-level IDS built herein, is data desynchronization. Thomas H. Ptacek, *et al.* [3], proved that the data received by a service and a network-based IDS, may be tampered with by means of insertion/evasion techniques. What this means is that the IDS will not receive the same data that the service does. Making application-level attacks impossible to detect. Another considerable problem is that these systems are of little or no use while trying to monitor encrypted information within a packages content.

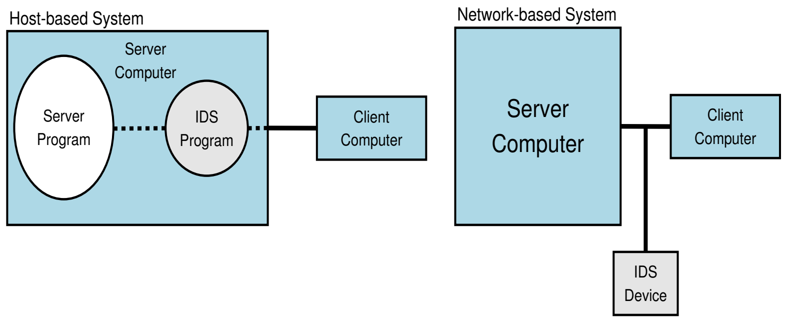


Figure 2.2: Difference between host-based and network-based intrusion detection.

Some network security policies try to find the compromise between using both types of IDSs. In this system layout, the various IDSs, will communicate with each other. This helps avoid the individual problems that each present.

2.1.5 IDS Fault Consequence Actions

What is meant by an IDS fault consequence action, is relevant to which actions will be taken once a fault occurs within the IDS. If the IDS comes to a faulty state, where it cannot continue to monitor events. Should the service that it is monitoring still be available or not? The two approaches to this issue, are fail-open and fail-close IDSs. Fail-open IDSs will leave the network open, accessible, after failure. This may be a security compromise, since once the IDS is brought down, the services are left vulnerable to intruders. Fail-close systems on the other hand, will take the services that it is protecting off-line after failure. This seems like the best approach, of the two, from a tight security point-of-view. The problem is that if the IDS is not implemented correctly, faulty situations may constantly arrive due to strange but benevolent network traffic. This will seriously affect the availability of the service.

2.2 Application-level Communication

Recalling the layered network architecture mentioned above, the application-level is the uppermost layer . This layer resembles the highest abstraction form of communication, where applications communicate by means of their protocols. An application's protocol defines; the sequence of actions that must be followed, in order to achieve successful data exchange between two or more entities following the same protocol. At this level of the layered architecture, applications communicate without notion of all data exchange happening in the inferior layers. For example, in an FTP transaction, the FTP-Server and clients are unaware of the TCP packages being sent to and from them. The server and clients just have notions of the FTP commands that they receive or send once a connection has been established. Connections between applications are handled at OS level. It is the OS's responsibility to manage the lower level communication of the network's architecture. Application commands are normally in the form of lines of strings (arrays of character sequences). The command's functional specification, are defined by the application protocol.

Once the concept of application-level data has been established, application-level vulnerabilities may be better understood. An example of what is not an application-level vulnerability, is the corrupt sequence of TCP packages trying to accomplish a Denial of Service (DoS) attack, at an OS buffer level. A good example of an application-level attack is when a certain character sequence is sent, with special meaning to the application. The character sequence might not mean anything at an OS level, but it may disrupt the applications normal behavior. For example, the stated character sequence might make the application return more information than it really should. These vulnerabilities are normally the result of application misconfiguration or application design or implementation errors.

Hence, application-level IDSs should analyze the final content that an application receives to its input. Not the bits and pieces that conform the process of sending and receiving data. As it may have been expected, application-level IDSs, are highly dependent on the functional specification of the protocol which the monitored application functions on. Although a strict IDS policy in analyzing just application-level content might prove to be a disadvantage at times. While analyzing application-data within a stateful context, a useful technique for marking request is by its source and destiny IP field. This data can be accurately obtained from the Internet layer. On the other hand, a useful technique for spotting distributed DoS attacks, is by marking a client's request by content, instead of its addressing. So a compromise should be deducted from what the IDS being designed shall be capable of detecting.

2.3 HTTP Application Data

A particular protocol, that has been a case of study for application-level IDS development, is HTTP (Hypertext Transfer Protocol) [11] i.e. relevant to Web-Servers and related applications. The reason behind giving such importance to a single protocol, is that Web-Services are normally, the main gateways between an organization's presence on the Internet and the general public. Hence, making all types of Web-Services access points to an organization's assets. Even if the organization is protected by gateway firewalls, Web-Services will normally be accessible from a DMZ[*] of an organization's computer network.

There was a survey conducted by the SecurityTracker team [10], during the period of April 2001 until March 2002. The total number of application software vulnerability alerts raised, during the establish period of time, were classified. From the total of raised alerts on application software vulnerabilities, 22.8% belonged to web-oriented applications. A total of 21 different categories of application alerts where identified, and web-oriented ones came in second place. A peculiar observation from this survey, is that the category of alerts that came in first place, at 25.9%, had no specific category. These where all the applications, that due to their functional specification could not be classified under a more general family. In third place, at 9.4 %, came alerts raised for security oriented applications e.g. firewall software and anti-virus software. These facts lead to believe, that really some of the most vulnerable services are those offered by Web-Servers and related applications. Having more time and effort being used, by private and

[*]The Internet is normally regarded as a DMZ.

public domain companies, for the development of Web-Servers and related applications then toward other types of applications. Hence, these applications are outnumbering the amount of applications within other categories.

It is important to differentiate between vulnerabilities and exploits that are identified in an application. Vulnerabilities are the cause of the faulty behavior, while an exploit is the actual effect (the faulty behavior). A classification is needed in order to efficiently take the appropriate countermeasures against intrusions. The classification can be for the various causes or the various effects. Sometimes within an application's vulnerabilities and exploits, the sets of causes and effects are associative. That is, a cause can cause several effects and an effect may be present due to several causes.

Next an underlining set of underlying HTTP vulnerabilities are presented. Due to the nature of the HTTP protocol, related vulnerabilities and exploits are associative. The set of HTTP related effects is inferior, in number, to that of the causes. There are two main effects that an intruder is after within a Web-Server or its related applications.

- Granting undesired/illegal access to a system and its data.
- Denying a certain service offered by a system.

If the classification is generated by effects, it would be too general. An outcome of this generalization is that the task of an analysis component (CIDF) of classifying becomes longer in time and inefficient. A less general classification method would be by the causes of the mentioned effects. This classification is divided into three classes of causes.

- Legal but undesirable activity within HTTP.
- Vulnerabilities within scripting languages.
- Illegal resource access.

2.3.1 Legal But Undesirable Activity

At times, there may be singular (outlandish) use of the HTTP protocol. A good example of outlandish use of the protocol is the hexadecimal encoding of characters. The intended purpose, that this feature was built into the protocol, is for special keyboards that do not have international language support. The vulnerability though, is that when a character or a sequence of them are encoded, a network-level IDS, might not detect any irregularity other than "arbitrary" data flowing to an application.

2.3.2 Scripting Languages

Vulnerabilities that are normally found between the Web-Server and the sub-system supporting the scripting language. They normally exploit some misconfiguration or a lower implementation error, such as buffer overflows. The scripting language that is most often targeted is CGI. Although misconfiguration problems are also a high target for intrusions. All scripting language sub-systems will normally present these misconfiguration vulnerabilities in their default installation configuration. For example, PHP , ASP , ASP.NET and Javascript . This general class of vulnerabilities in scripting sub-systems, can be broken up into more specific classes, for example:

- SQL injections, where a user will have partial or complete access to undisclosed information. This is achieved by passing faulty SQL commands to a scripting sub-system.
- Cross-Site scripting, where an intruder can obtain information from a third party. That is, another user using the breached Web-Server.
- Buffer overflows, where a user will be able to execute commands with the privileges of the Web-Server. At the same time this vulnerability grants a user access to undisclosed information. This vulnerability is strongly related to the next category of vulnerabilities.

2.3.3 Illegal Resource Access

This type of vulnerability causes can be summarized in three main categories. These categories are presented with some more specific examples.

1. Access failure to protected resources are caused by:
   - Path manipulation, where an intruder may be trying to obtain an overview of the system's file structure, and map-out installed vulnerable web-applications.
   - Password guessing, where an intruder may be trying to access undisclosed information.
   - Overloading the system and generating a DoS scenario.
2. Access failures to inexistent resources are caused by the same vulnerabilities as above, except for password guessing.
   - User guessing, where an intruder will try to map existing user-names in the system by guessing a Web-Server's home directories. For example, in the Apache Web-Server these would be directories which precede the user-name with a tilde,

$$http://servername/\sim username$$

3. Access failure to server error-prone resources. That is, resources that cause faulty behavior within the server. These are caused by the same vulnerabilities as accesses to inexistent resources.

Chapter 3

System Analysis

3.1 Overview

After having presented the domain description, there is a broader idea of the environment in which the IDS will perform its activities. In this chapter, existing systems are analyzed to gain knowledge of what kind of functionality fully deployed application-level IDSs, monitoring HTTP traffic, provide. The knowledge gathered from these existing systems along with the domain description can then be used to elicit system requirements. These requirements, as will later be presented, are broken up into two categories; all required functionality towards an end-user and that which the IDS must meet as a system, to perform in its environment.

3.2 Existing Systems

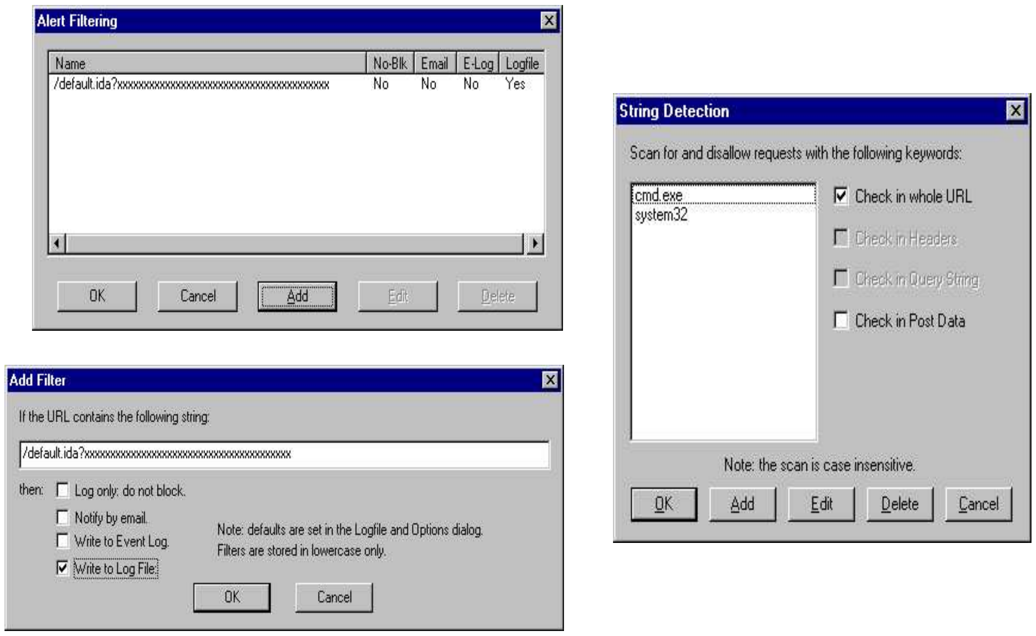
From the existing systems presented in this document, there are three main families. Those that function under strict pattern matching, rule-based systems. Those systems that try to correlated stateless events within an HTTP request with the OS system calls that the Web-Server will make during the processing of the request. More advanced systems that function under pattern matching as well, but where patterns are not elicited from rules (regular expressions), but request features i.e. Neural Network-based classification systems. The last group of systems, are those that are oriented towards a pure stateful analysis of incoming and outgoing HTTP traffic events.

3.2.1 Titan

The Titan software system is an application developed and commercialized by Flicks Software [15]. This tool is a signature-based stateless IDS. It behaves like a firewall wrapped around Microsoft IIS. Titan may be categorized as a host-based system, since in its specifications it must be installed in the same computer where the web-server resides.

This tool behaves like a firewall in that through its user-friendly GUI, figure 3.1[*]. The system administrator specifies rules that the IDS will look for in an HTTP request. The signatures

[*]The figure was obtained from [15].


Alert Filtering

Name	No-Blk	Email	E-Log	Logfile
/default.ida?xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	No	No	No	Yes

Add Filter

If the URL contains the following string:

/default.ida?xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

then: ☐ Log only: do not block.
☐ Notify by email.
☐ Write to Event Log.
☑ Write to Log File.

Note: defaults are set in the Logfile and Options dialog.
Filters are stored in lowercase only.

String Detection

Scan for and disallow requests with the following keywords:

cmd.exe
system32

☑ Check in whole URL
☐ Check in Headers
☐ Check in Query String
☐ Check in Post Data

Note: the scan is case insensitive.

Figure 3.1: String detection and filter description GUI window of Titan.

that Titan looks for are defined strings which if found tick off the alarms and relevant counter measures taken. The evaluated version of Titan, does not support the description of regular expressions . Another system that behaves in the alike of Titan i.e. a firewall-like system, is Apache's Mod_Security [16]. This system provides more flexibility in describing rules since it supports searching for POSIX defined regular expression. This system is embedded into the Apache Web-Server. A sample configuration can be seen in figure 3.2 [†].

From this sample configuration file, there are three sample rules listed. The first rule helps in preventing XSS tampering of Web-Sites. It checks all incoming variables for JavaScript allowing only the HTML variable to have such values. The second rule is used against port scans, searching for the existence of the HTTP User-Agent and Host header-fields. If these are not present the request is dropped. The third rule specifies that all dropped rules (deny) shall be logged and responded with a 500 Server-Error Status-Code response.

The inconvenience with both of these host-based systems, is that they do not keep track of certain events that have occurred *i.e.* they do not perform stateless inspection. It is left up to the administrator to be clever enough to realize that a sophisticated attack is taking place based on the logs generated by these systems. The fact that these systems are host-based, do not make them optimal for a heavily loaded Web-Server. Although pattern matching does not consume many resources, these systems may affect the performance of a server within the order of tens of thousand hits per second e.g. one of Google.com's Web-Servers.

[†]These sample instructions were obtained from [16].

3.2.2 SecureIIS

This tool is within the family of IDSs along with AppliCure TotalShield [19] and McAfee Entercept Web-Server Edition [17]. These tools add-on an extra feature to that of signature-based stateless inspection of HTTP Request. They correlate the system calls made to the OS during an HTTP transaction. These systems do perform some sort of stateful analysis in that a certain set of established system calls are recognized as an intrusion. An example of this type of behavior is presented in figure 3.3[‡].

```
<IfModule mod_security.c>
        ...
     SecFilter "ARGS|!ARG_html" "<[:space:]*script"
     SecFilterSelective "HTTP_USER_AGENT|HTTP_HOST" "^$"
     SecFilterDefaultAction "deny,log,status:500"
        ...
</IfModule>
```

Figure 3.2: Sample instructions from Apache's configuration file Mod_Security section.

SecureIIS [14] is embedded into Microsoft IIS, as AppliCure TotalShield is into the Apache web-server. McAfee's Entercept can be embedded into these two Web-Servers as well as into iPlanet's Web-Server. All three provide an advanced friendly GUI, through which an administrator can customize the behavior of the system. These systems are host-based, not only embedding themselves between the WWW client-server communication but also in between the server and the OS. These systems have a considerable impact on the a Web-Server's performance, which makes the suitable for a small to medium sized Web-Server in terms of hits per second.

These systems are fail-close systems, meaning that if the IDS is compromised it will bring the Web-Server completely out of service. Not only disallowing access to the the Web-Server from the DMZ, but disallowing any sort of Web-Server access to system resources such as; databases and file-systems. These systems, in relation with their modus operandi and localization, are preemptive. Although most of them may be switched on sniffer mode. Sniffer mode, is used for logging purposes, where data is used after an intrusion has taken place. This analysis technique is considered as forensics examination, and is the common mode that most NIDSs operate under.

3.2.3 Intelliwall

The system that seems to be fitted best for large-scale Web-Servers and has one of the more general classification techniques is Bee-ware's Intelliwall [12]. This network-based IDS, sits on a specialized device either under sniffer mode or serving as a firewall-like system 3.4[§]. Intelliwall is the only system, from the ones mentioned, that includes neural network technology for

[‡]The figure was obtained from [17].

[§]The figure was obtained from [12]

classifying anomal and misuse HTTP requests. The system also presents a GUI through which a system administrator can interact with it.

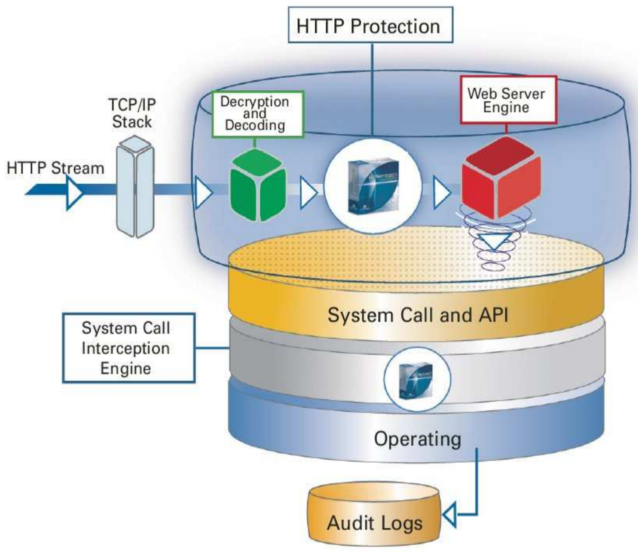


Figure 3.3: Entercept's interaction with the Web-Server and its environment.

This is system is highly portable since it can be used with any Web-Server. It does not affect the performance of the Web-Server since all computations are done on its intended hardware device. This system differentiates between anomal and misuse events, by the extraction of certain features from all incoming events. This property makes the system a very powerful classification tool. The only feature that Intelliwall does not present is that it does not correlate sequences of events. It "learns" to accurately detect stateless events, by having two lists of predefined anomal and misuse events. But from the specifications presented in [12], it does not correlate client requests with server responses. This technique allows a system to model client behavior.

3.2.4 WebSTAT

This last tool, is built around the STAT framework [18]. The STAT framework is an intrusion detection set of tools that operate at different levels of the networks architecture, and then intercommunicate to reach accurate conclusions. This system is based on automaton models of misuse scenarios which are hard coded into a the application-level IDS. More than WebSTAT itself, the STAT framework is a powerful utility around which IDSs can be built. The only inconvenience with this framework is its complexity and its poor adaptability to future trends of net traffic once put into applications.

The reason behind this poor adaptability is that the scenarios, modelling stateful inspection, are hard coded into the system. WebSTAT is a host-based system which is embedded into into the Apache Web-Server. It has a surprising low impact effect on the Web-Server's performance. In the average case affecting the total HTTP transaction time by less than 0.5% of its original

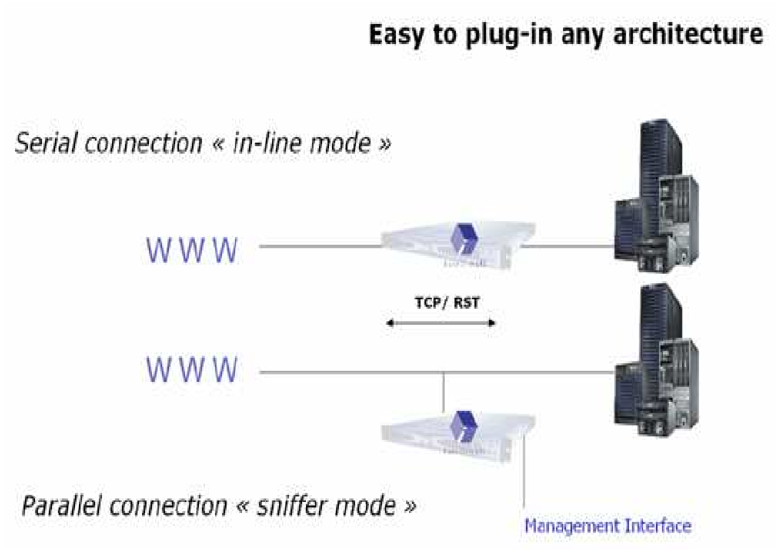

Easy to plug-in any architecture
Serial connection « in-line mode »
WWW
TCP/ RST
WWW
Parallel connection « sniffer mode »
Management Interface

Figure 3.4: Intelliwall's device architecture. This device may be switched between a networks sniffer to a firewall-like specialized device.

time.

3.3 Comparisons

Most of the system comparisons could only be made at a conceptual level. In table 3.1 the most relevant IDS paradigms are presented from each of the above mentioned systems. Where there is a reference to the "switch" value, it means that the IDS can switch between one value or the other mentioned. For example, Intelliwall can function as a preemptive device or as a passive analysis tool. From the systems shown, non present true stateful analysis of HTTP traffic, except for WebSTAT. The type of stateful analysis is based on the correlation of HTTP requests and a Web-Server's resource usage (system calls). The false-positive rate of these systems could not be tested on all. It was only tested on Mod_Security, and Titan since most of these tools are withing the private domain.

Most information gathered for those systems that could not be tested, was from their corresponding white-papers (data-sheets). From which concrete data could only be extracted for Intelliwall. One of the most interesting features from this system is the amount of requests that it can process per second, twenty five thousand. From the systems that were tested, Titan and Mod_Security, they function in a similar manner. Both systems presented a high level of false-positives since they are based on string-based regular expression matching algorithms. Although all well known attack were acknowledged. It should also be noted that from the systems that do present stateful inspection (AppliCure, Entercept and SecureIIS), it is not a stateful inspection correlating Web-Server response status-codes with requests. It is through the correlation of system calls and client requests. This helps in preventing critical future intrusions, i.e. buffer overruns (BoF). But is of not much use for detecting events implicit to the HTTP protocol itself.

For example, the indirect disclosure of restricted information.

	Localization	**Counter**	**Analysis**	**Fail Mode**	**Embedded**
Intelliwall	network-based	switch	stateless	switch	own device
WebSTAT	host-based	passive	stateful	fail-open	Apache
SecureIIS	host-based	switch	stateful	fail-close	IIS
Titan	host-based	switch	stateless	fail-close	IIS
Mod_Security	host-based	preemptive	stateless	fail-close	Apache
Entercept	host-based	preemptive	stateful	fail-close	Apache, IIS and iPlanet
Applicure	host-based	preemptive	stateful	fail-close	Apache

Table 3.1: Conceptual comparison between some HTTP Application-level IDSs.

All of the systems except Mod_Security presented GUIs as an interface to the system. Mod_Security's user interface is through a sequence of instructions present in the Apache configuration file. Those systems that did present GUIs, make user interaction more intuitive than that of Mod_Security's. The way that the system communicates events to the user is done through log file in Mod_Security. While on other systems they may be either log files or GUI notifications.

3.4 Neural Network Classification

The technology that should be used for this system is based on neural networks. As opposed to that of traditional pattern matching-based categorization systems, neural network-based categorization systems present a higher level of scalability, adaptability and accuracy. The reason behind these enhancements is that neural-network systems have the ability to be trained to recognize related sets of data. The recognition process with neural-networks is not done by traditional pattern matching. It is done by finding features representative of the data that needs to be classified. A draw-back with neural-network technology is that finding representative features is a time consuming process. Which is later paid back in terms of recognition speed and accuracy.

The use of neural-network algorithms for an intrusion detector is ideal since the IDS is really a classification system. The power that is delivered by classifying suspicious events by domain related features, provides accuracy levels that are not delivered by ordinary pattern matching systems. The scalability that neural network systems provide, from classifying between small sets of data to large ones do not bring loss of accuracy to the system.

The adaptability that neural network system present is fundamental under the HTTP domain. Web-applications vary greatly in content, as mentioned earlier in this document. What may be classified as misuse behavior for one type of web-application might be sane content to another. Therefore, with the training ability that neural network systems present, they can be adapted to almost any environment. Again it their training ability that also makes them less sensible to changes in future trends of requests. Neural networks can be retrained or instances of new significant training data added to what the system already knows.

Neural Networks also have another advantage over conventional pattern matching algorithms. Neural Networks are not sensible to noise, i.e. these algorithms can recognize given patterns even when the exact training pattern is not available. Unlike pattern matching algorithms that can only provide exact matches of their coded patterns.

3.4.1 Available Algorithms

There are a wide variety of algorithms implementing neural network theory available. These can be classified by different aspects. One of these being how the neural network is trained. The type of training is considered either supervised or un-supervised . Given that a basic notion on neural network theory is understood, supervised training is where the algorithm is fed not only a set of inputs but also an associated set of outputs. Unsupervised learning on the other hand, the algorithms are only provided with stimulus i.e. input data. The algorithms job is to find, given certain criteria within the networks internal setup, different classes of categories within the input data.

Associating a set of inputs with known outputs is what makes neural network algorithms of interest to the given domain. It is the wide variety of web-application data-flows that have a meaning to one application and a different one to another. Therefore, an algorithm implementing supervised training is required, in order to customize the IDS for the Web-Server's context of operation.

	Training set accuracy	**Test set accuracy**	**Training time (s)**	**Test time (s)**
FANNC	100%	100%	523	1
Fuzzy ARTMAP	100%	99.1%	537	1
CPM	100%	62.9%	435	4,179
SuperSAB	95.5%	94.7%	6,923	9

Table 3.2: Neural Network algorithm performance comparison during the Telling-Two-Spirals apart test.

From the available algorithms that where reviewed, there were representatives of ART (Fuzzy ARTMAP), from Field Theory (CPM), a mixture of ART and Field Theory (FANNC) and a representative from Back-propagation (SuperSAB) [6]. From an experiment performed on all four families of algorithms Telling-Two-Spirals-Apart Zhihua Zhou *et al.* [6] , the results are displayed on table 3.2[¶]. From these results it may be seen that the FANNC algorithm is not only one of the most accurate ones during testing but also one of the fastest. Considering the tight real-time constraints that the IDS, described herein, must meet during an HTTP client-server transaction. In the following sub-section, the FANNC algorithm is over-viewed.

3.4.2 Selected Algorithm

The selected algorithm is FANNC (a Fast Adaptive Neural Network Classifier). This algorithm is based on supervised learning and is and requires one-pass training to achieve high predictive accuracy. FANNC is also characteristic for its high learning as well as testing speed. The neural network is divided into four layers: an input layer, an internal classification layer, an external

[¶]This results table was obtained from [6].

classification layer and an output layer. Initially the algorithms hidden layers are empty and as new data arrives units are adaptively appended.

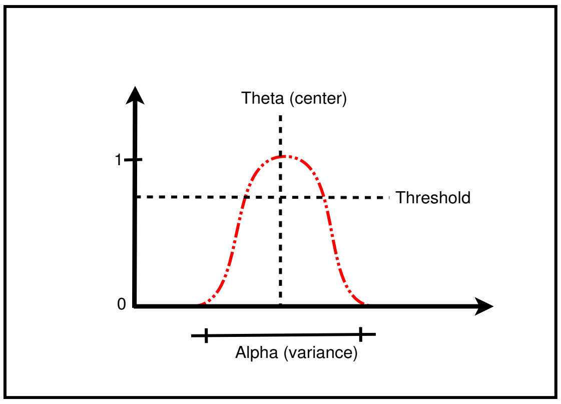


Figure 3.5: FANNCs Gaussian weight parameters.

The idea behind the FANNC algorithm is that it starts creating multi-dimensional basins out of each training instance of data fed to the system. Each training instance contains *n* input units. Each input unit is modelled with a Gaussian weight , figure 3.5. As the learning process begins the basins representing data input will be modified. Their width and center are modified as similar instances of data are arrive. If the data is too different, and cannot be covered by the current basin, new ones are inserted. For a detailed description on how the algorithm is designed refer to 3.5.

3.5 Requirements Elicitation

The following sub-sections will discuss, in a structured manner, the requirements that the HTTP IDS must fulfill. These requirements range from a user's perspective to the functionality that the HTTP IDS must provide as a black-box system. The elicitation process has been done through the knowledge gained from; the problem's domain description and functionalities that existing systems provide. Note that, the requirements are itemized without a particular notation e.g. PDL language. No PDL was used due to flexibility within the elicitation process. It has proved to be less time-consuming to describe and modify the requirements with natural language rather than with a PDL.

3.5.1 System Requirements

System requirements are broken up into three categories: the system's localization, its dependence on a particular Web-Server or OS and its behavioral models. Behavioral models describe the type of analysis the system should be capable of performing. In response to analysis how the system will react under positive and negative findings. Another aspect of the systems behavior is its fail-mode i.e. how should the system react under an internal faulty state. These system requirements are described in the following sub-sections.

3.5.1.1 Localization

The system should act as a host-based IDS being physically localized as a network-based IDS. The system should benefit from both network and host-based advantages. The IDS should have a specific hardware device assigned to it. The hardware device may range from a stand-alone computer to a customized hardware device. The IDS should behave, by default, like a reverse-proxy.

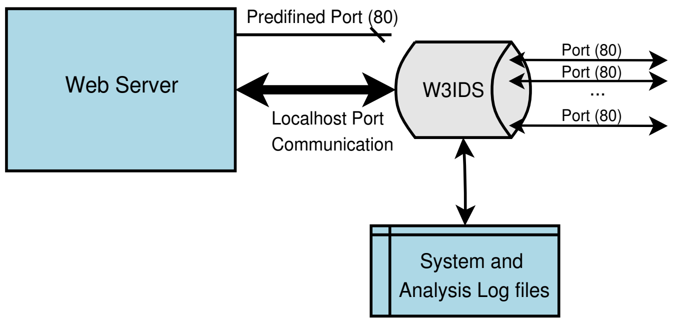


Figure 3.6: Reverse proxy like IDS filtering all traffic from and to clients.

In a reverse-proxy setup 3.6, the IDS is place in between the clients and the servers. All traffic is buffered in the reverse-proxy, in this case analyzed and then forwarded to its destiny. Notice that NIDS evasion/insertion techniques do not apply under a reverse-proxy setup. All information that the reverse-proxy receives it forwards to its intended destiny. Therefore, clients cannot connect directly to the servers avoiding data desynchronization. A reverse-proxy scheme adds another indirect level of protection to the servers. It hides, at a transport layer-level, their IP addresses making public only that of the reverse-proxy's.

3.5.1.2 Portability

The system should not be dependant on any particular Web-Server. It should be implemented as an independent entity which analyzes traffic outside of a Web-Servers API. As mentioned in the previous requirement it is implemented as an external entity, the reverse-proxy setup, to the Web-Server. This allows the IDS to coexist with a wide variety of HTTP/1.1 compliant Web-Servers. The IDS residing on a different hardware device other than the Web-Server, opens a wide range of possibilities of OSs for the IDS to run on.

3.5.1.3 Behavior

- **System Analysis -** The system should be capable of combining a stateless form with a stateful form of analysis. The stateless form of analysis should be used to identify suspicious and potentially harmful individual client requests. During stateless analysis the neural networks' classification capabilities are exploited. After these have been trained with representative data of a suspicious activity's features, they should raise the proper alarms during testing. User defined suspicious activity features should be elicited with the help of auxiliary tools. These tools will be presented in the following chapters.

If an alarm is raised, the proper actions should take place in order to commence stateful analysis. Stateful analysis should only start, if the features extracted from the request do not form part of a user defined feature blacklist. The system should record before stateful analysis starts: the clients IP address, the type of alarm that its suspicious request has raised and the server responses that it received. With the correlation of these three data-types, the system should search for user-defined temporal patterns. For example, if a stateless alarm has been raised, by the same client IP address, receiving more than fifty server response 404 Client-Error status codes take relevant countermeasures. Significant temporal patterns should also be obtained through auxiliary tools.

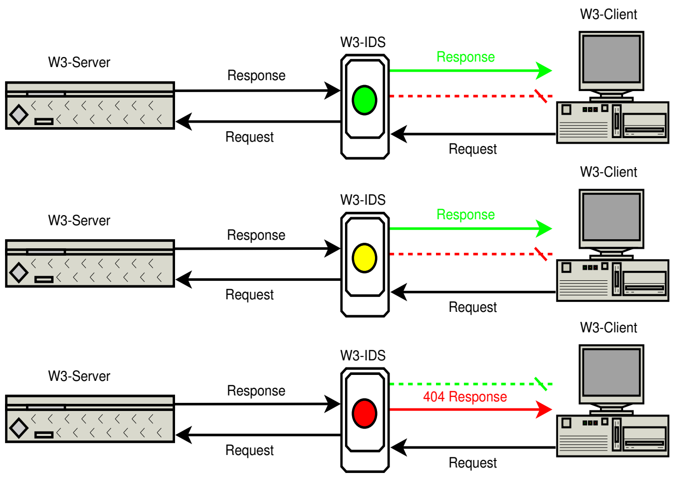


Figure 3.7: Use-case scenarios for the countermeasures sub-system.

For the temporal patterns to be of use, these must not only classify events by client IP addresses but also by the types of stateless alarms raised. This form of classification will provide data necessary to avoid several types of distributed intrusions. This is where the term *source entity* is introduced. A *source entity* contains a client IP address, an alarm number and a Web-Server associated response Status-Code. These two properties characterize the clients request under the current environment. The system must have notions to keep track of all suspicious activity by interpreting suspicious requests as source entities. For a better understanding of this concept refer to the kernel data-structures section in chapter 4.

- **System Countermeasures -** The type of countermeasures that the system takes are two types; the logging of events and interception of data-flow. That is, if a request is found to meet critical criteria it is filtered and the client should then receive a decoy response. In this case a 404 Client-Error Status-Code instead of a more informative response. It has been shown that informative responses are giveaways of the existence of an application-level IDS somewhere in-between the Web-Server and the clients. In most cases, resulting as an invitation for the intruder to instigate more on the system's network layout. In order to detect temporal patterns, the system must forward those request that are only found suspicious but not critical. This behavior is modelled in figure 3.7.

If a request is found to be critical (some of its features belong to the blacklist) or a temporal pattern has been met, the decoy response is not sent immediately. A delay is associated to the source entity, which grows in an exponential manner as the the source entity's item (an IP address or alarm number) insists. The system's exponential response delay behavior is shown in figure 3.8. In order to avoid a DoS against the system itself, their should be a user-defined maximum delayed time. This maximum response delay time is a system parameter. After the source entity's item has reached its maximum delay it should be marked as a forbidden source entity item. In which case, an immediate 404 status code response is sent back and the connection to the client computer closed.

- **System Fail-Mode -** It would not make much sense for the system to be fail-open, if it was going to be used for more than a passive informative analysis tool. Due to the systems reverse-proxy requirement, if the system encounters an undesired state, the Web-Servers are taken off-line. Fail-close systems have a critical draw-back inherent to their design. If they are poorly designed and implemented, they become an indirect form of DoS towards anomal server requests.

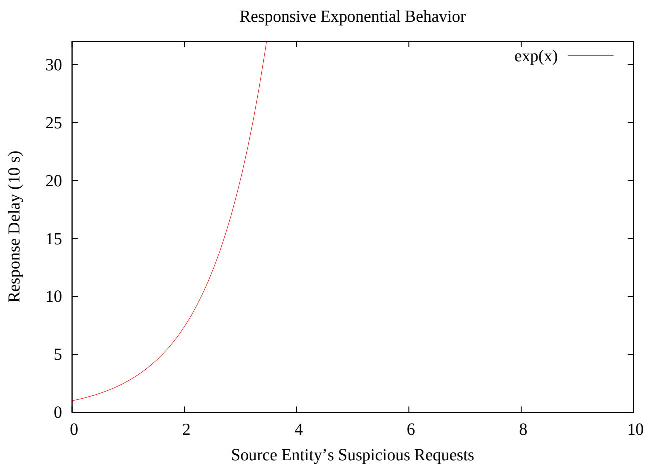


Figure 3.8: If a source entity insists with suspicious requests, the system exponentially increases the response delay. Every unit of delay is worth ten seconds.

3.5.2 User Requirements

The first step in eliciting user requirements is establishing a user profile for the system. The system is aimed at an experienced system/network administrator. The system administrator should have average knowledge of the HTTP protocol. At least to the point of knowing what each of the standard message headers, methods and response-status codes mean. The interaction between the system and user, should be at a system configuration level and internal data status reports. These requirements are described as follows:

1. The user must be provided with some means of interacting with the system. The interaction should allow the user to modify certain aspects of the IDSs behavior. The interface should also allow a user to obtain knowledge on the system's internal data status. The interface should be provided in the form of either a configuration file and a graphical or console-based user interface.

2. The configuration file shall help the user define some of the system's parameters at system startup. For example, the ports where the system will listen for client requests or the number of neural networks used to classify. This configuration file shall have a basic text format. It should also have a set of well defined options that the user can specify. An example file can be seen in appendix B, and its available options will be described in the following chapters.

3. Given the neural network-based nature of the IDS, the configuration file should have a section where the user can describe request features. These features extract relevant HTTP request information, which is then fed to the neural networks for content inspection. For example, a request's feature could be the content size of an HTTP request header. The presence of the particular header can also be regarded as a request's feature. These features should be described in a structured manner, for example, boolean or arithmetical expressions.

4. The configuration file should provide a section where the user can specify the neural network training data. This data should be provided in the form of individual files. Each file containing the training input that representing a class of suspicious behavior.

5. The configuration file should provide a section where a user can define a feature blacklist. This blacklist allows the user to describe well known request features *e.g.* the presence of the `"/etc/passwd"` or `"DROP%00TABLE%00*"` strings in the URL content. These features should also be described in a structured manner by the user.

6. The configuration file should provide a section where a user can specify temporal patterns. These temporal patterns define the correlation of client requests with their responses. These correlations help a user identify certain forms of intrusions and information disclosure. These temporal patterns shall be specified by the user in a structured manner. The user should specify these temporal patterns as boolean expressions.

7. The user interface, whether it be graphical or console-based shall be of an informational manner. Later as will be discussed in chapter 7, this behavior should be modified for IDS management purposes. The user interface should present, in a formatted manner, the status of the system's internal data-structures. A good option for presenting data is the XML markup language. XML could allow the access of system information remotely, from a web-based environment, in a secure manner.

8. In addition to the user interface, the system shall provide the user with a logging facility. The logging facility shall record events in real-time regarding not only request analysis. It should also record relevant system events, for example, the events occurring during system startup and run-down. These logs, should help the user during analysis providing data generated by the request analysis process. System logs provide the user with information during forensics examination. This information is useful, in the event that the IDS reaches an undesired state it cannot recover from.

Chapter 4

System Design

4.1 Overview

Throughout the system design, all conceptual ideas for implementing the ideas proposed in chapter 3 are presented. These are either explained with pseudo-code, described or depicted. The depictions include commonly used techniques such as: layered architecture diagrams, control-flow diagrams, data-flow diagrams and client-server architecture diagrams.

Design is divided into three main section. The first one enumerating the necessary components that will be required during system implementation. This part of system design is also considered as logical design. The principle technique used during logical design was brainstorming. The second section explains in detail each of the components identified during the brainstorming phase. The third and last section, describes how all system functional components interact with system data structures. The interaction between functional components is also described.

During brainstorming, functional components are sub-divided into four categories. The first category involves the system's back-bone functional components. The relevant data-structures form part of the system's back-bone. Due to the neural network nature of the system, the next three categories involve actions that are required during: system start-up, training and testing.

After logical design, specialized depictions are used for components identified during brainstorming. These are mainly the architecture-descriptive diagrams mentioned above. For example, how the IDS's kernel is organized or what components form the system's data-structures.

In last section system component interaction is depicted mainly by control-flow diagrams. The diverse algorithms used are, in some cases, described with pseudo-code.

4.2 Logical Design

This section mentions most of the components that the IDS shall need during its design. All system and user requirement gathered in chapter 3, are used as a foundation during brainstorming. There are two main classes of logical components; system data-structures and functional components. Within the following diagrams, all data-structures have a light gray background color while processes are depicted with a violet background color. Since the IDS is based on neural network technology, functional components are split-up into three types. All components required upon system startup, during system training and system testing.

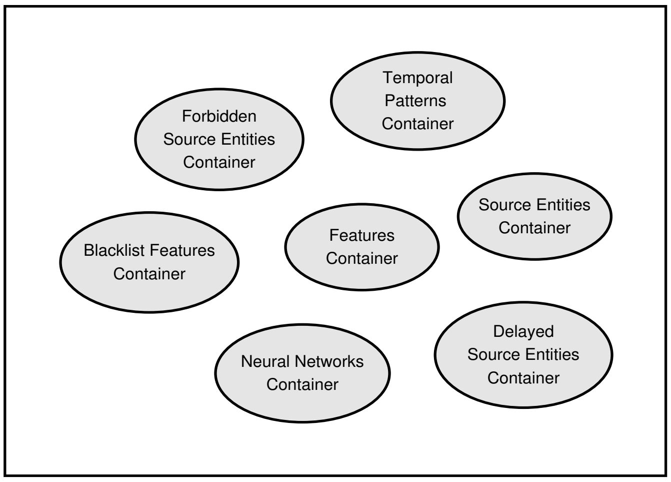

Temporal Patterns Container
Forbidden Source Entities Container
Source Entities Container
Features Container
Blacklist Features Container
Delayed Source Entities Container
Neural Networks Container

Figure 4.1: Main system data-structures components.

4.2.1 Required Data-structures

The system's data-structures, figure 4.1, are sub-divided into two groups. The first group is composed of dynamic while the second are static system data-structures . The first group, are all of those data-structures that are modified in size during run-time. These data-structures are the *Source Entity Container*, the *Delayed Source Entity Items Container* and the *Forbidden Source Entity Items Container*. The static group of system data-structures are those that are created upon system start-up. These data-structures will not be modified during run-time, they remain as read-only containers. The remaining data-structures, the *Features Container*, the *Blacklist Features Container*, the *Temporal Patterns Container* and the *Neural Networks Container*, form the static group.

4.2.2 Required Functional Components

In figure 4.2 the system's back-bone functional components are presented. These are: the *IDS's Kernel* , the *Connection Manager* , the *Service Request CIDF Threads* , the *Manager Console* and the *Logging Sub-system* .

The main functional components, required during system start-up, may be seen in figure 4.3. Start-up is characterized by the data-structure initializers, both static and dynamic. After the system loads all of its parameters with the *Kernel Parameter Extractor* , it begins all initializations. Initialization includes generating the static data-structures i.e. the *Neural Network Container*, the *Features Container*, the *Black-list features Container* and the *Temporal Patterns Container*. It includes the system reserving memory for dynamic data- structures. And reserving all required network sockets from the OS and initializing them. Notice that the *Neural Network*

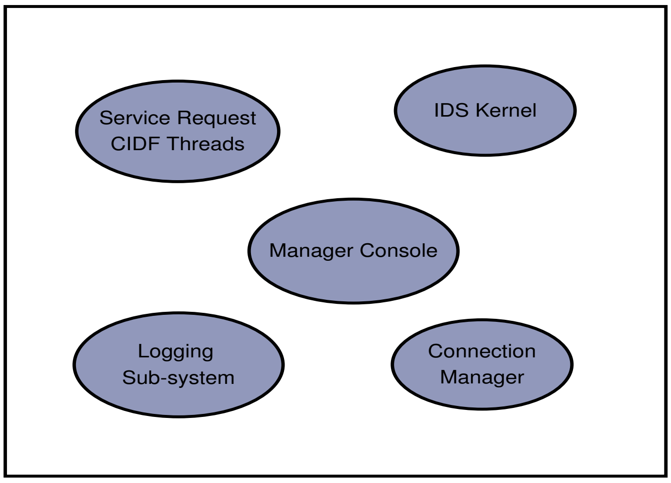


Figure 4.2: Main system components including data-structures.

Trainer is presented like a main component during start-up. It is not really a major component, since it forms part of the *Static Data-structures Initializer* . That is why it is out-lined with a dotted line, unlike the others. The importance of this component, is remarked from trying to divide the systems behavior into three phases: a start-up, training and testing phase.

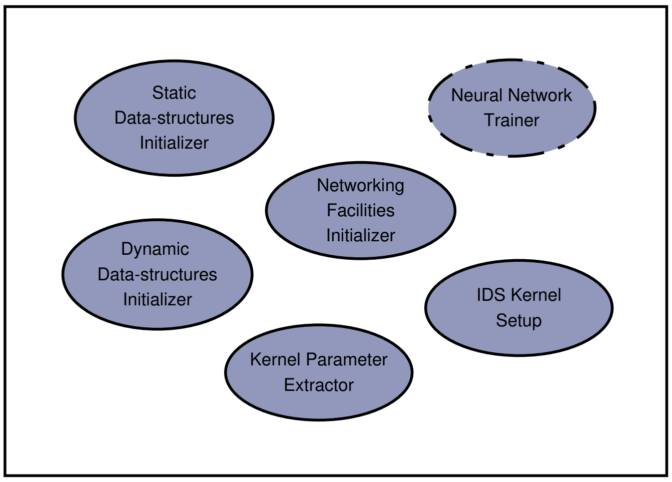


Figure 4.3: System components required during system startup.

Considering future system enhancements, the abstraction of an *Neural Network Trainer* functional component should be considered. Simplifying the task of the neural networks on-line training. From figure 4.4 the functional components required for training are presented. The *Train Data Fetcher* is in charge of fetching data from specified files. Given the functional specification of the FANNC library, developed in parallel with this project, data must be scaled between zero and one before it is fed to the neural networks. Therefore, a *Train Data Scaler* is required. For the actual training, of the objects within the *Neural Network Container*, a series of threads are required. To measure system performance, a form of temporal measurement is required during the training phase. These are the *Temporal Cost Measurement Utilities* .

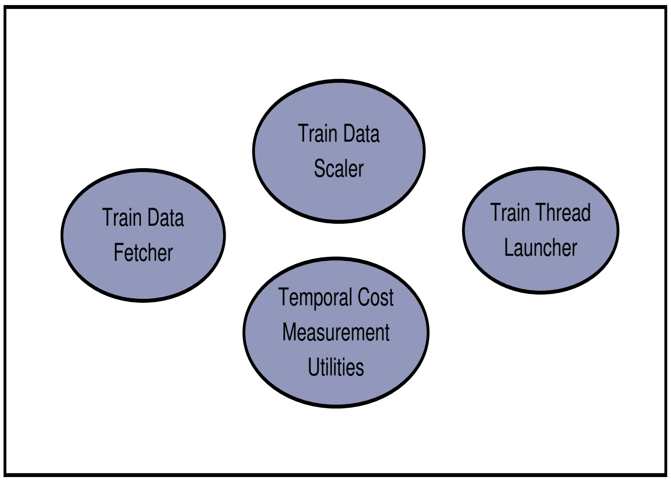


Figure 4.4: System components required during training of the kernel's neural networks.

The core components of the IDS is the functional components required during system testing. These components model a modified version of the architecture proposed by CIDF. The main component within system testing is a *Connection Manager Initializer*. This component is in charge of launching a *Service Request CIDF Thread* each time a client requests a connection. The components within a CIDF Thread are, the *Events Box*, the *Analysis Box* , the *Counter Box* and the *Storage Box*. In order to measure the temporal cost efficiency of the system, certain *Measurement Utilities* are needed.

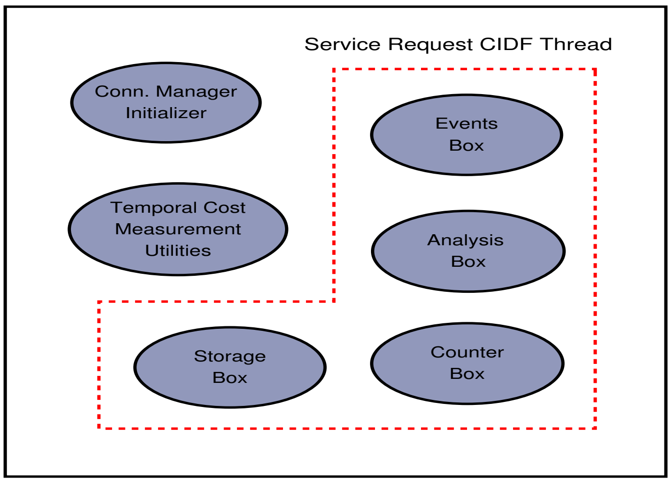

Service Request CIDF Thread
Conn. Manager Initializer
Events Box
Temporal Cost Measurement Utilities
Analysis Box
Storage Box
Counter Box

Figure 4.5: Functional components required during system testing.

As with system training, while using the neural networks for testing, multiple threads are used. Therefore, the measurement utilities help in evaluating several aspects of the testing process. For example, the average time in which the system is capable of handling an HTTP client-server transaction. Other relevant evaluation parameter, are the time in which the neural networks perform their analysis and the time required to evaluate temporal patterns.

4.3 System Architecture

The components that were mentioned during the logical design phase are now described in detail. The required system data-structures are broken down into data that will be need to meet the established requirements. How the data-structures are organized will also be presented. Functional components will be broken down into sub-processes that intercommunicate. This process decomposition will be presented with the help of data-flow diagrams. Not only will sub-process interconnections be shown, also the data exchange among them.

The following conventions are used for the data-flow diagrams presented in the following sub-section. All input/output entities are presented as slanted boxes with a light violet background color. All functional components are presented as circles with a light blue background color. All internal system data is presented as a formatted box with a light gray background color. The tagged arrows indicate the direction of component data-flow.

4.3.1 Kernel

The IDS's kernel, within an object-oriented abstraction, is an object that holds the main functional components. It also holds the system's data-structures, providing the functional components an access to these data-structures. The IDS's kernel provides the Manager Console with an interface through which information can be extracted. The MCI (Manager Console Interface), figure 4.6, is a set of defined methods. Through these methods, the Manager Console extracts statistical as well as data-structure status information from the Kernel. The Kernel is responsible for the sequence of start-up events, this includes the training of its neural networks.

If on-line training mode was supported by the kernel, it would be responsible for switching from training mode to testing mode and vice-versa. Through the MCI, the kernel would receive the relevant command to make the testing phase stop. After all testing components were stopped the kernel would proceed with the training phase.

During the training phase the kernel is in charge of all neural network threads spawned. The kernel holds an internal container with each of the thread IDs. This internal ID list is required so that the kernel knows when all of the threads have finished. After which the kernel can either start or switch back testing mode.

4.3.1.1 Manager Console Interface

The kernel's MCI is characterized by a set of methods that are publicly available to the Manager Console wrapper, figure 4.6. These methods are:

- A method to extract the status of the neural networks internal state.

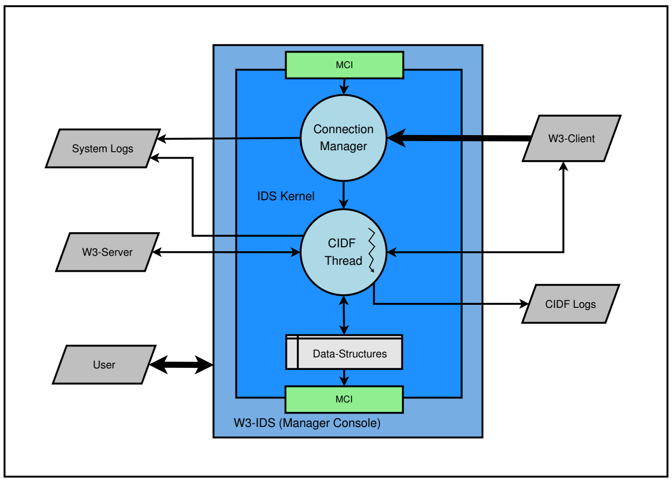


Figure 4.6: The Kernel's layered architecture.

- A method to extract the status of the Source Entity Container.
- A method to extract the status of the Delayed Source Entities Container.
- A method to extract the status of the Forbidden Source Entities container.
- If on-line training is supported, a method to notify the connection manager that training is about to begin. Hence the Connection Manager should pause all of its activity.
- A method to indicate the kernel that the IDS is shutting down. Therefore all allocated memory should be freed and all live threads either joined or killed.

These set of methods are the only components publicly available to the Console Manager wrapper. There for the CMI remains for informational purposes only and shutting the system down.

4.3.2 Data-structures

System data-structures are divided into two classes; the static and dynamic data-structures. The main difference, as mentioned earlier, is that static data-structures are created during system start-up. These do not grow in size as the system begins its analysis process. Whereas the dynamic data-structures are initially empty. As suspicious events and forbidden source entities are found during the analysis process, these data-structures increase in size. Notice that if on-line training was supported, the Neural Network Container would have to be a dynamic data-structure instead. Since the data-structure components would increase in size each time training instances were added.

Both dynamic and static data-structures must be protected with semaphores during system testing. Multiple CIDF Threads will solicit to access the data-structures' components. Since this project has been developed an object-oriented technique; several threads could ask, at the same time, to execute one of the data-structures element's methods cause inevitable a faulty state within the system. This particular problem could have serious effects when computing the Neural Networks' output.

4.3.2.1 Static Data-structures

These are the features container, the blacklist features container, the temporal patterns container and the neural networks container. These data-structures are generated during system start-up and are customized with the system parameters read from the configuration file. Both the features and blacklist features containers are closely related to the neural networks container. The size of the both the features containers determine the amount of input that each component within the neural networks container will have.

- **Features Container -** The features container is set of user-defined features which are specified in the configuration file, appendix B. These features model the properties that wish to be extracted from each client HTTP request. The features container is a vector that contains as many components as have been specified. Each component contains a data element which is the string representing the feature and a method under the function domain as shown bellow. Each component may be seen as a parser engine. This extract method requires a special HTTP request object and the string line resembling the feature. These are the features that are used as training and testing data for the neural networks.

  $$\boxed{\begin{array}{rcl} features & : & String; \\ extract & : & String\ x\ HTTPRequest \rightarrow \mathbf{R}; \end{array}}$$

  Notice one of the parser engine arguments is of type HTTPRequest. An HTTP message parser was written to fulfill the requirements of this project. The HTTP parser receives as input a single String or several of them. Then it tries to structure the contents of its string into HTTP jargon. Its contents can then be easily accessed by other components making use of it. The parser supports both types of HTTP messages i.e. requests and responses. The full specification of how this additional library is used can be found in appendix (HTTP Parser).

  Feature definition makes use of *FDL* (Feature Description Language), developed for this project. The extract member method implements the parser engine for FDL. With *FDL* a user can declare features, within the configuration file, in a structured manner. The description language allows the user to declare certain boolean and arithmetic expressions that model features. Its complete specification is presented in the following section. An example FDL expression is; a user-declared feature that models the size of the User-Agent field in a request. The syntax for this expression would be: *header.User-Agent.size*. FDL models several types of features that may be extracted from a request, *e.g.* if a request-field contains a defined regular expression, the occurrences of a string within a request-field or the presence of certain request-fields. In section (Behavioral Models), how the feature extraction process is carried out will be described in detail. It is through FDL, that the user can customize the security policy of the application-level IDS.

- **Black-list Container -** these features are also declared using FDL . These are treated in a different manner by the system. Only FDL boolean expressions may be declared as black-list features. An example black-list feature is:

  header.Cookie.regex(< html javascript*)*

  The above feature is an FDL boolean expression. When this feature is extracted if the regular expression, enclosed within the parenthesis, is found in the Cookie header-field the *extract* method returns its interpretation of true. Boolean values are interpreted as real numbers; a one for true and zero for false.

- **Temporal Patterns Container -** The temporal patterns container, is analogous to the features container. It is also a vector that contains objects which represent temporal patterns. Each component of this vector has the same structure as those of the features container. Except that the *extract* method, the parser engine, receives different parameters and behaves in a different manner.

  This container is used to find meaningful data between the correlation of suspicious source entities and server responses. The source entities container will be described in the following sub-section, along with the dynamic data- structures.

tmp_pattern	:	*String;*
extract	:	$String\ x\ SRCENTContainer\ x\ SRCENTItem \rightarrow$ **boolean**;

  Another description language was developed in order to search for user-defined temporal patterns within the source entities data-structure. This is the TPDL (Temporal Patterns Description Language) and this description language is also presented in the following section. TPDL is composed solely of boolean expressions. As can be seen from the parser engines arguments, it receives a user-defined temporal pattern, the source entity bin and the data modelling a suspicious request. For example, a user could specify that a suspicious chain of events was characteristic by an IP address that had tipped off the same alarm more then fifty times. Receiving each time a Server-Error response (*5xx*). This temporal pattern is declared with the following instruction;

  NET.Server-Error.5xx > 50

  Another example instead of searching by indexed IP addresses, in order to recognize a distributed attack, perform the search by indexed alarms that have been tipped off by certain addresses. Receiving in each case more than seventy Client-Error (*4xx*) status-codes within the server response. This temporal pattern is declared with the following instruction;

  IP.Client-Error.4xx > 70

4.3.2.2 Dynamic Data-structures

These are all of the data-structures that are initially empty, and as analysis is carried out begin to increase in size. These include the *Source Entity Container*, the forbidden source entity items container and the delayed source entity items container. A fundamental building block of the dynamic data-structures is the Source Entity abstraction.

alarm_id	:	*Integer*
ipaddr	:	*String*
status_code	:	*String*

The *alarm_id* is used to tag the alarm that went off. Neural Networks are abstracted as alarms, and the one which generates the highest output value, is interpret as a sounding alarm. Output is generated from the input request features. If none of the neural networks generate output greater than zero, then none of the alarms have been activated. The *ipaddr* data member, holds the IP address of the client that tipped off one of the alarms. The *stats_code* data member contains the value of the status-code within the server response. Notice that if a request is considered suspicious but not critical it will be forwarded to the server.

All of the dynamic data-structures hold two types of indexes. In order to associate alarm IDs, IP addresses and status-code responses, there must always be one index for suspicious IP addresses and another for suspicious alarms set off. This is clarified in the following data-structure descriptions. A relevant data-type abstraction used for all dynamic data-structures is that of an associative map. These are also commonly known as hash-tables and the concept is shown in figure 4.7.

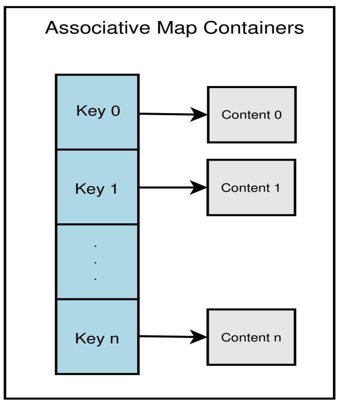


Figure 4.7: Associative maps data-type abstraction.

Hash-tables help accelerate lookup time, since elements are not indexed by position. Elements are indexed by unique keys, therefore lookups are performed with constant temporal cost $O(T) = const$, while with other traditional data-structures used, for example vectors, lookup times grow depending on the containers size i.e. a lineal search temporal cost $O(T) = n$, where n resembles the size of the vector.

- **Source Entities Container -** this data-structure should be used to store two sub-containers. Both of these sub-containers store suspicious requests along with their source host IP addresses and the associated server responses. The difference is that one data-structure shall be indexed by host IP addresses and the other by suspicious requests. Two different sub-containers are required to avoid vulnerabilities against distributed attacks. That is why an IP address that originates a suspicious request as well as the suspicious request shall be categorized as source entities. As was described in the previous sub-sections, temporal patterns can be specified for either IP addresses or suspicious a number of times that an alarm has been set off. To get an idea of the layout, these data-structures are depicted in figure 4.8.

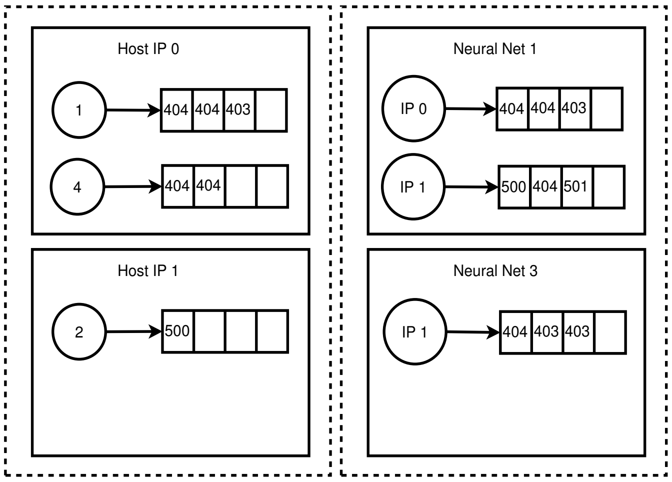


Figure 4.8: Required data-types of the Source Entities Container.

The advantage of having two different sub-containers is that the system shall not be vulnerable to distributed attacks, where IP addresses are spoofed. The system can categorize by both host IP addresses and by the output of the neural networks. The only disadvantage will be the temporal cost of lookups in both sub-containers for temporal patterns. Suspicious requests are enumerated by the number of the neural network that has recognized the request as suspicious.

Each element in the IP indexed sub-container is defined as:

ipaddr	:	*String*
alarm_ents	:	*Map(Integer x Vector(String))*

Client IP addresses are interpreted as Strings. The *alarm_ents* is an associative map (Hash table), where each key is a unique alarm id and its associated content a vector with corresponding server responses. This data abstraction relates to IP addresses the alarms that they have sounded off. With each alarm the status-code that the server responded with. A vector is used because an IP address can sound off the same alarm several time in a given time frame. Each time the server response status-code must be recorded in order to successfully implement the temporal patterns concept.

Each element in the Alarm indexed sub-container is defined as:

alarm_n	:	*String*
ip_ents	:	*Map(String x Vector(String))*

In this sub-container the way that data is related is by which addresses have sounded off an alarm, and each time what server response status-code did those IP address get. Notice that both of these sub-containers are associative. All data that is contained in one will be in the other. Except that it will be ordered (interpreted) in a different manner.

- **Delayed Source Entity Items Container -** This data-structure also contains two sub-containers. One to index source entity items by IP addresses and another by alarm number. This data-structure is used to keep track of those source entity items that should receive a delayed *404* Client-Error status-code server decoy response. A source entity item that meets this requirement may be because it is an IP address that has sent a request that contained features from the blacklist or eventually met one of the user-defined temporal patterns. The source entity item may also be an alarm number that has met one of the user-defined temporal patterns. That is, if alarm is on the delayed list because some clients have tipped it off, and it has met one of the user-defined temporal patterns. When a new client connects and it sounds off this delayed alarm. The new client will receive a delayed decoy response according to the temporal delay associated with the alarm.

delayed_ips	:	*Map(String x Double)*
delayed_alarms	:	*Map(Integer x Double)*

The *delayed_ips* data member is a hash-table which uses as keys IP addresses. The *delayed_alarms* data member uses as keys alarm numbers. The associated *double floating-point* number with each key, is a delay time. This is the decoy response delay that should be enforced as a counter-measure against critical Source Entities. The *delayed_alarms* uses as keys the alarm numbers.

- **Forbidden Source Entity Item Container -** This data-structure follows the same sub-division, as the other two dynamic data-structures. The forbidden entity items container is used to place all of those delayed source entity items that have met their maximum user-defined delay time. The reason behind this data-structure is to avoid an DoS attack against the IDS itself. This DoS attack is described in the following sub-section.

forbidden_ips	:	*Map(String x Integer)*
forbidden_alarms	:	*Map(Integer x Integer)*

The *forbidden_ips* data member is a hash-table which uses as keys IP addresses. The *forbidden_alarms* data member uses as keys alarm numbers. The associated The associated *Integer* value with each key is the number of times that a forbidden source entity item has attempted to access the system, in case its an IP address. In case its a forbidden alarm number, the number of times that it has been sounded off. A source entity item is placed in this container if it has met its maximum delay time in the Delayed Source Entity Items. This behavior models the handling of a persistent HTTP client request, either by origin (IP address) or because it has sounded an alarm too many times. With exponential increasing response delays, an intruder is highly probable to become disinterested in their persisting with faulty requests. There are check-ups during the testing control-flow. In these check-ups if a clients associated source entities are forbidden, a *404* Client-Error status-code is immediately sent and the client connection is closed.

4.3.3 Connection Manager

The connection manager is in charge of listening for client connections. This component retrieves all of its networking information from the kernel's parameters. For example, where the hidden web-server resides or what port to listen for client connection requests at. When a client connection request arrives, the connection manager, launches a CIDF thread to service it. The data-flow from the connection manager to the CIDF threads is presented in figure 4.9. The connection manager must give each service thread the address of all system data-structures. These are wrapped up in an object, a main class hierarchy diagram is presented in chapter 5. It must provide the thread with the file descriptor of the granted clients connection. It must provide the service thread with the whereabouts of the Web-Server.

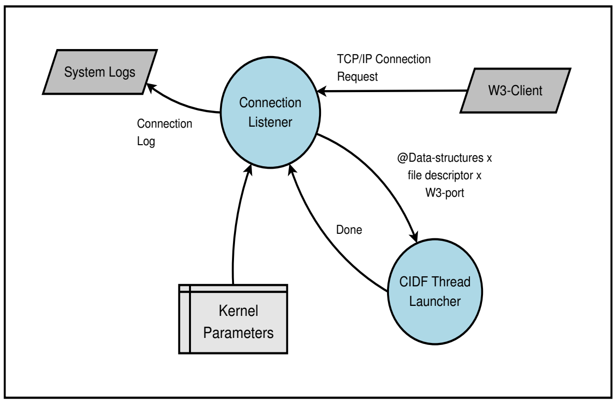


Figure 4.9: Connection manager's data-flow context diagram.

Internally the connection manager also keeps certain data-structures to measure temporal parameters. Auxiliary OS dependent libraries have been used, these are discussed in chapter 5. These temporal parameters are extracted by the kernel's MCI for their visualization. The total client service request time is measured, the amount of connections established as well as a averages of both of these parameters. Another function of the Connection Manager is to log any relevant activity in the system log files. The connection logs are a sequence of formatted strings that are written immediately to the system log file.

The connection manager does not make use of a thread pool for the launched CIDF Threads. If the IDS is on a specific hardware device, the use of a thread pool would be desired. Using a thread pool with the type of countermeasures that the system carries out, may lead to a DoS attack against the Connection Manager. This DoS attack is depicted in figure 4.10. The scenario implies that the intruder Joe knows that there is an application-level IDS between him and the Web-Server. He also knows the internal workings of the IDS. The intruder could potentially fill up the thread pool with threads, that must response with a certain delay time. This would deny service to Mary, who is not an intruder. She could expect to wait from her arrival, at most the user-defined maximum delay time. When the temporal constraint is met, one of the CIDF Threads servicing one of Joe's request will service Mary. This DoS attack is not a critical threat, since the IDS is not completely taken off-line. Although it is rather annoying for the regular users of the Web-Servers behind the IDS.

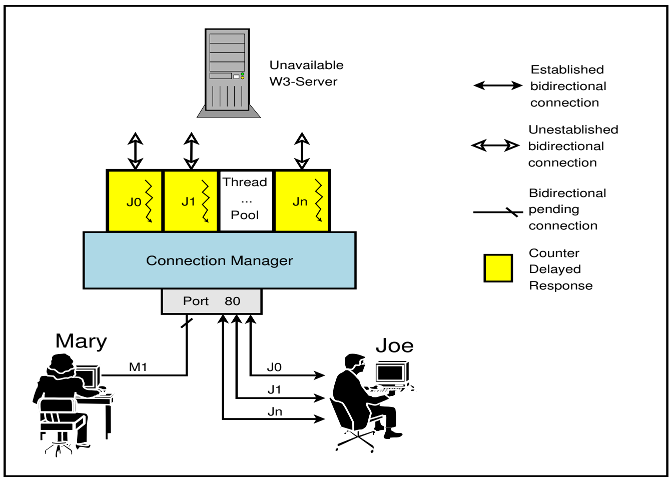

Unavailable W3-Server
Established bidirectional connection
Unestablished bidirectional connection
Bidirectional pending connection
Counter Delayed Response
J0
J1
Thread ... Pool
Jn
Connection Manager
Port 80
Mary
M1
Joe
J0
J1
Jn

Figure 4.10: Scenario for a DoS attack against the system's Connection Manager.

In order to solve this issue, each time a client connection request arrives, a new thread is created. The disadvantage with this approach, is that the amount of required system resources is not known until they are needed. That is, how much dynamic memory is used depends on the amount of threads that are alive during a certain time frame. Joe under this approach could, potentially, congest the Connection Manager with threads. This scenario also causes a DoS, but in a more retarded manner. An efficient fix to this problem will be presented in chapter 7

4.3.4 CIDF Threads

The CIDF framework proposed in chapter 2 is not followed in a rigorous manner. It has been modified to fit the requirements of the IDS and to simplify design. The data-flow between its major components is presented in figure 4.11. Each thread has semaphore protected access to the kernel's data-structures. The abstraction of a Countermeasure Box as well as the Storage

Box are designed as CIDF Thread member functions. These are called from within the Analysis or Events Box. The data exchange between the Analysis and Events Box is only a boolean flag, “continue”, which will determines if analysis is required or not.

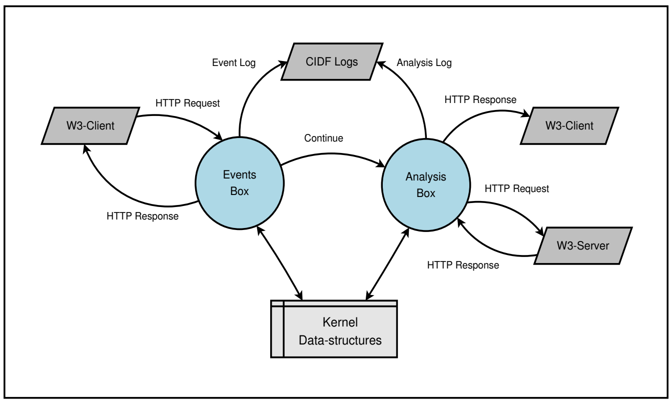


Figure 4.11: CIDF Thread’s data-flow context diagram.

The CIDF Threads have two different types of logs. This are related to the events box or the analysis box. The both logs are a sequence of formatted strings that are written immediately to the CIDF log file.

4.3.5 Logging Sub-system

The abstraction of a sub-system is designed as a shared function of CIDF Threads, the Connection Manager and the kernel. The function domain is described as:

$$\boxed{\textit{log\_event} \quad : \quad \textit{String x String)} \rightarrow \text{Boolean}}$$

The first parameter to the function is the file-name where the log is to be placed. The second parameter is the log content. Each component that calls the event is in charge of internally formatting the string to a suitable format. A suitable format includes: the time and date of the log, the component that is calling the log_event function and the actual event that has triggered the logging activity. The partial contents of the system log file is displayed in figure 4.12.

There are several levels of priority within log messages. These can be either informative or critical. This abstraction between two types of messages will help the user. This aid can be used during forensics examinations, due to the IDS reaching a faulty state it cannot recover from. The other form of logs are those generated by the CIDF threads. These are analysis oriented whether they come from the Analysis or Events Box. The partial contents of the CIDF log file is displayed in figure 4.13.

4.3.6 Manager Console

The Manager Console, as presented in figure 4.6, is a wrapper around the IDSs kernel. This wrapper has access to a defined set of kernel member methods, the MCI. Provided with the

```
[15/Jan/2005:11:17:03 +0100] kernel msg: System started.
[15/Jan/2005:11:17:03 +0100] kernel msg: Training neural networks.
[15/Jan/2005:11:17:03 +0100] kernel msg: Connection Manager started.
...
[15/Jan/2005:11:17:03 +0100] Conn msg: Incoming request 127.0.0.1
[15/Jan/2005:11:17:03 +0100] Conn error: Client closed connection.

[15/Jan/2005:11:17:03 +0100] Console error: Could not fetch delayed.
```

Figure 4.12: Partial content of the system related log file.

```
127.0.0.1 [15/Jan/2005:11:17:03 +0100] Events msg: Delayed black-list.
127.0.0.1 [15/Jan/2005:11:17:03 +0100] Analysis msg: Insert in forbidden.
...
130.225.9.1 [15/Jan/2005:11:17:03 +0100] Analysis msg: Alarm 4 sounded off.
```

Figure 4.13: Partial content of the system related log file.

system's API, the MCI allows any type of informative program to be wrapped around the kernel. The manager console developed here in is based on a system standard output. It allows for the user to execute a set of basic commands on the W3-IDS after it has been started. As mentioned before, these are of an exclusive informative nature. The set of commands that a user can execute are:

- System run-down.
- Data-structure status retrieval.
- System Temporal Statistics.

The implemented FANNC library returns the state of a selected neural network in an XML formatted string. For the time, the CMI does not format the information it extracts to XML. It returns its own format which may only be visualized through the standard C++ Input-output Stream libraries. An example of a users interaction may is presented in the console output from figure 4.14

In this Manager Console output, the user solicits to view the contents of the dynamic system data-structures.

```
=== Displaying IP index table ===
IP Addr: 127.0.0.1
Alarm: 0 -> 401 401 401 401 401 401 401 401 401 401 401 401 401 401 401 401
IP Addr: 130.225.137.12
Alarm: 0 -> 403 403 403 403 403 403 403 403 403 403 403 403 403 403 403 403
IP Addr: 98.32.11.1
Alarm: 0 -> 404 404 404 404 404 404
=================================
=== Displaying Net index table ===
Net Alarm: 0
IP: 127.0.0.1 -> 401 401 401 401 401 401 401 401 401 401 401 401 401 401 401
IP: 130.225.137.12 -> 403 403 403 403 403 403 403 403 403 403 403 403 403 403
IP: 98.32.11.1 -> 404 404 404 404 404 404
==================================
=== Displaying delayed IP source entities ===
=== Displaying delayed Alarm source entities ===
Alarm: 0, 4
=== Displaying forbidden IP source entities ===
IP: 98.32.11.1, 1
=== Displaying forbidden Alarm source entities ===
```

Figure 4.14: Console output of user interaction, showing a user fetching the Dynamic Data-Structures' content.

4.4 Description Languages

The expression languages described in this section, allow the user to set the security policy of the IDS. That is, by defining what the IDS should look for within a client request. The user also defines what type of patterns of behavior the system shall search for within the correlation of client IP addresses with alarm numbers and the respective server response status-code. Both description languages share the same design principle.

They are recursive sub-systems, based on arithmetical and boolean expressions computation. Recursion allows the user to specify complex as well as atomic expressions. A complex expression requires the the parser engine to perform actual arithmetical calculations or boolean comparisons. While atomic expressions may be either variables or numbers. Variables must be looked-up depending on which one of the languages the expression is being defined for.

The languages are defined using a BNF. Both languages are strongly typed against the HTTP version 1.1 specified in [11]. Example expressions are provided with each language specification.

4.4.1 Feature Description Language

The FDL (Feature Description Language) is used to specify user-defined features. These features must be extracted from an incoming HTTP client request. Then, they are used to characterize the request passing the characteristic request features to the system's neural networks.

The following BNF is used to specify the FDL:

```
    feat_aexp  ::=  Real | Variable | feat_aexp + feat_aexp
                    | feat_aexp - feat_aexp | feat_aexp * feat_aexp
                    | feat_aexp / feat_aexp
    feat_bexp  ::=  True | False | feat_aexp = feat_aexp
                    | feat_aexp < feat_aexp | feat_aexp > feaet_aexp
                    | not feat_bexp | feat_bexp and feat_bexp
                    | feat_bexp or feat_bexp
     Variable  ::=  message-line.section.feature
 message-line  ::=  request-line | header | body
      section  ::=  method | url | version | Host | User-Agent
                    | Content-Length | . . .
      feature       type | size | regex | occurrence | IDEN
```

In order to specify nested expressions, parenthesis are needed to give an order of priority. About the above specification, if the message-line is the body it has no section. The suspensive dots in the “section” element, refer to general-header, entity-headers, request-headers described in [11]. The function domain for each of the feature extracting functions are:

$$\begin{aligned} \textit{size} &: \textit{String} \rightarrow \mathbf{R} \\ \textit{regex} &: \textit{String} \rightarrow \textit{Boolean} \\ \textit{occurrence} &: \textit{String} \rightarrow \mathbf{R} \end{aligned}$$

The size feature returns the content size of a message-line section. The regex feature is provided with a regular expression, and returns true or false if a match is found. The interpretation of Boolean values, within the given context, is a 1.0 or a 0.0. The occurrence feature behaves in a similar manner to the regex feature. It is provided with a regular expression and then searches for the number of times the pattern appears in the content of the message-line section. The IDEN function is the identity function. This feature indicates if a give message-line section is present or not. The type feature will enumerate the available methods, specified in [11], and return the code value of an HTTP request method. Some example expressions are:

```
request-line.url.size
request-line.method.type
header.Cookie.IDEN and (header.Cookie.occurrences(%00) > 0)
header.User-Agent.size
```

4.4.2 Temporal Patterns Description Language

The TPDL (Temporal Patterns Description Language is used to specify user-defined temporal patterns. These temporal patterns are searched for in the Source Entity Container. Temporal patterns may be defined for the two components that characterize a suspicious client HTTP request. These are either by an HTTP request origin i.e. a client IP address, or by the stateless alarm that the HTTP request sounded off i.e. a neural network ID. Temporal patterns are described by the correlation of both elements of a source entity with the corresponding server response status-code. There is one main difference between this description language and FDL. The TPDL's parser engine does not make sense out of arithmetical expressions. These may only be used within boolean expressions. The following BNF is used to specify the TPDL:

```
          tp_aexp  ::=  Integer | Variable | tp_aexp + tp_aexp
                        | tp_aexp - tp_aexp | tp_aexp * tp_aexp
                        | tp_aexp / tp_aexp
          tp_bexp  ::=  True | False | tp_aexp = tp_aexp
                        | tp_aexp < tp_aexp | tp_aexp > feaet_aexp
                        | not tp_bexp | tp_bexp and tp_bexp
                        | tp_bexp or tp_bexp
         Variable  ::=  Source-Entity.Status-Code-Type.Status-Code
    Source-Entity  ::=  IP | NETID
 Status-Code-Type  ::=  Informational | Success | Redirection
                        | Client-Error | Server-Error
      Status-Code       1xx | 2xx | 3xx | 4xx | 5xx
                        | 100 | 101
                        | 200 | ... | 206
                        | 300 | ... | 305 | 307
                        | 400 | ... | 417
                        | 500 | ... | 505
```

When the parser engine does variable look-ups, it first defines what kind of source entity is the temporal pattern defined for. If IP is defined, then the lookups in the Source Entity Container will be done in the sub-container that is indexed by IP addresses. If NETID is defined, the lookups are done in the sub-container indexed by neural network IDs (alarm number). As may be seen from the language's specification, there are five types of server response status-codes. Notice, as stated before, in the sub-container indexed by IP addresses the alarms that that IP address has sounded off are registered in the hash-table. The alarm number is used as the key and it is unique. Each time this scenario is present, the corresponding server response status-code is inserted in the vector associated with the key. Here are some example expressions that the TPDL parser engine can interpret.

```
(IP.Client-Error.404 > 30) and (IP.Redirection.3xx < 5)
NETID.Client-Error.404 > 25
(IP.Server-Error.5xx > 15) or (NETID.Success.2xx > 0)
```

4.5 Behavioral Models

The content of this section is mainly in a graphical manner. This section contains depictions of the control-flow diagrams of major system components. These are mainly what activities are carried out during system testing. Inter-callings on other system activities is also provided as a form of control-flow, e.g. what activities are carried out during system start-up and the training of the neural networks. Notice that the inter-callings lay-out does not fully reflect the system's implementation. The inter-callings only reflect in a compact manner which activities should be carried out.

4.5.1 System Start-up Activities

During system start-up mainly memory is allocated for the required data-structures. The static data-structures are created and memory is allocated for memory that might be required for the dynamic data-structures. As part of start-up activities, the training of the system's neural networks is carried out. The following inter-callings describe the sequence of events that happen during system start-up.

```
init_mode()
   L_ set_kparameters(‘‘.conf file’’)
   L_ init_kdata_structs(STATIC)
     L__ load(FEATURES)
     L__ load(BLACKLIST)
     L__ load(T_PATTERNS)
     L__ load(N_NETWORKS)
          L___ train_mode()
   L_ init_kdata_structs(DYNAMIC)
     L__ reserve_mem(SRC_ENTS)
     L__ reserve_mem(DELAYED)
     L__ reserve_mem(FORBIDDEN)
```

The load functions push-back, through the user-defined kernel parameters, the corresponding objects in their vectors. When loading the neural networks, not only is each object inserted into its container, but after the system switches to training mode. The reserve_mem functions only create a reference to a dynamic data-structure. Which will begin to have objects inserted as the testing of HTTP requests is carried out. If the system should fail during any of its initializations, memory allocation or during the training process, it exits providing relevant logs in the system log file.

4.5.2 System Training Activities

From the inter-callings shown bellow, notice the that the get_kparameters receives TRAIN_DAT_LIST which represents the training_data_list from the configuration file, appendix B. Keeping in mind that the kernel holds internal data-structures to keep track of time, it uses init_cronos and stop_cronos. These functions should time the amount of time that it takes each thread to train its assigned network, and the time which it takes all threads to carry out their task. Other relevant statistical values that may be obtained from timings are the averages of each.

```
train_mode()
    L_ get_kparameters(TRAIN_DAT_LIST)
    L_ fetch_data()
    L_ scale_data()
    L_ init_cronos(GNL_TIMING)
    L_ launch_threads()
             L__ init_cronos(NET_TIMING)
             L__ feed_nnetwork()
             L__ stop_cronos(NET_TIMING)
    L_ join_threads()
    L_ stop_cronos(GNL_TIMING)
```

When the feed_nnetwork() inter-calling is carried out, the following pseudo-code describes how the threads are launched, and what each thread does. Notice that for a detailed description of how the FANNC library is used, refer to appendix C.

Pseudo-code presenting how the neural network training threads ara launched

```
$P = \{ neural_nets.size \neq 0 \wedge$
    $start : @FANNCNetwork \rightarrow void\}$

$thread\_vec$ : $\mathbf{Array}[0..neural\_nets.size]$ $\mathbf{of}$ $FANNCThread$;
$\mathbf{for}$ $i$ := 0 $\mathbf{to}$ $neural\_nets.size$ $\mathbf{do}$
    $thread\_vec[i].start(@neural\_nets[i])$

$\mathbf{for}$ $i$ := 0 $\mathbf{to}$ $neural\_nets.size$ $\mathbf{do}$
    $thread\_vec[i].join()$

$Q = \{\}$
```

Pseudo-code presenting hot the neural network objects are trained.

```
$P = \{ train\_input.size = train\_output.size\ land$
    $train\_input.size \neq 0 \wedge net : @FANNCNetwork\}$

$train\_data$ : $\mathbf{Array}[0..instances]\mathbf{of}(in : Array[]\mathbf{Double},$
                                    $out : Array[]\mathbf{Double});$

$\mathbf{for}$ $i$ := 0 $\mathbf{to}$ $train\_data.size$ $\mathbf{do}$
    $net \rightarrow train(train\_data.in[i], train\_data.out[i]);$

$Q = \{\}$
```

Notice how the “@” and “→” symbols are used to differentiate between argument passing. This is commonly known as accessing a variable by value or by reference. In this case when the special operators are used, it means that the variable is accessed by reference “@”. When the contents of the variable are accessed, if it is a data-structure, the “→” symbol is used.

4.5.3 System Testing Activities

Upon a client connection request to the IDS, the Connection Manager launches a CIDF Thread with. The Connection Manager provides the CIDF Thread, with a file descriptor that indicates the connection medium to the client e.g. a TCP/IP Berkeley socket. During the system testing the first set of relevant activities are that of the CIDF Thread. These activities are presented in the control-flow diagram in figure 4.15. The series of events here, differ from those proposed in the original CIDF framework. The original proposal is just used as an abstraction, for comprehensive purposes. The Storage and Countermeasures Box are called from within the Events and Analysis Box as member methods of the CIDF Thread class. The Events Box returns a boolean value, “Continue” to the threads main execution flow. This boolean variable indicates if the Events box has not already taken relevant countermeasures against a client request.

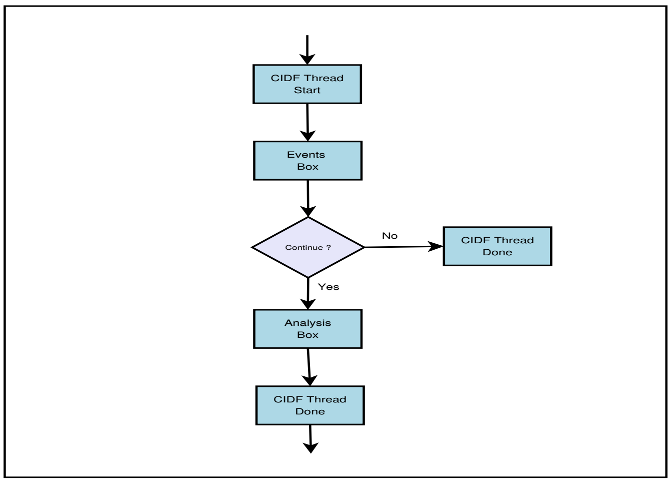


Figure 4.15: Control-flow diagram for the CIDF Thread.

As the diagrams are broken down, the activities within the Events and Analysis Box are presented. The Events box, figure 4.16 is in charge of checking that the serviced client request is not in the forbidden IPs list. It is also in charge of verifying that a client HTTP request has no black-list features and of extracting its characteristic features.

The feature extraction process may be seen in figure 4.17.

The process of feature extraction starts by receiving client input. It then parses the received data into an HTTP. In figure 4.17 the method for starting the FDL parser engine is presented. The same manner is used to extract black-list features from the request. Once the Events box is finished, if no countermeasures have been taken against the client request, the analysis may start. Notice that countermeasures are only fired, within the events box, if a request presents

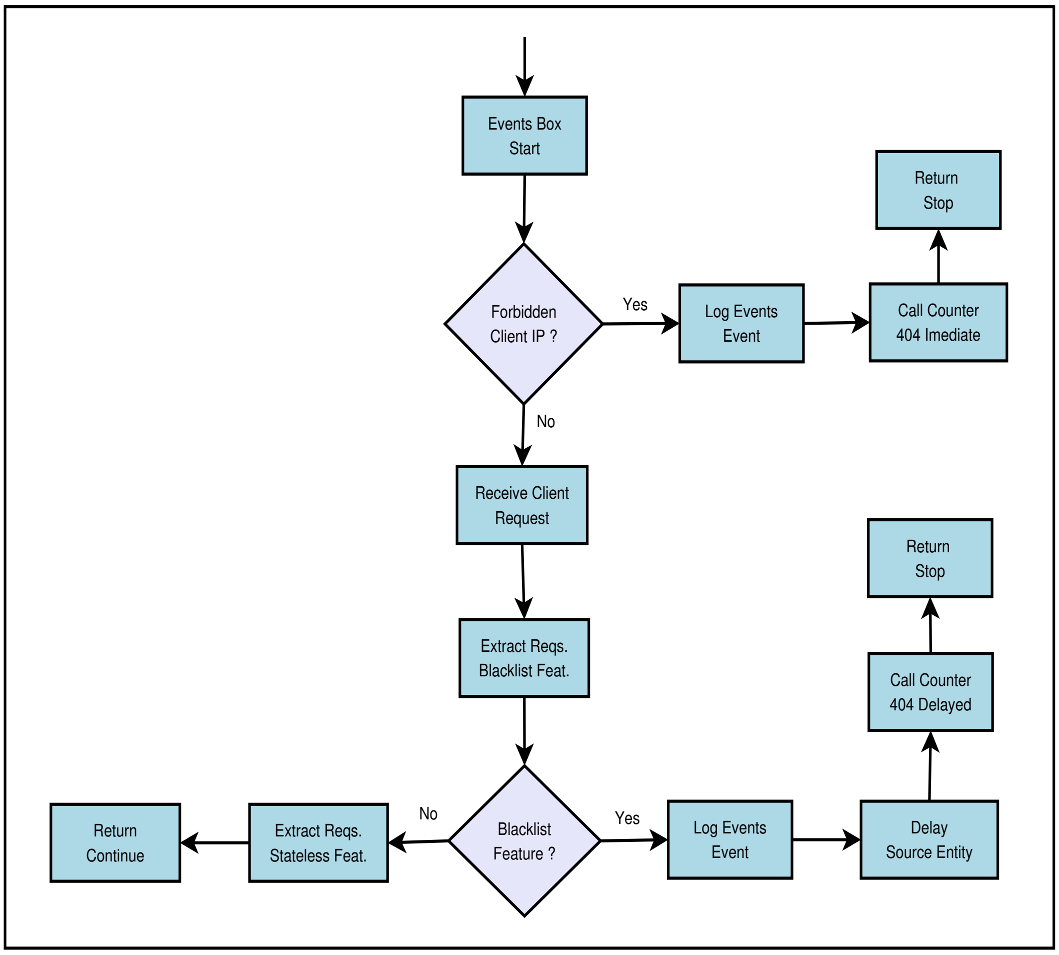
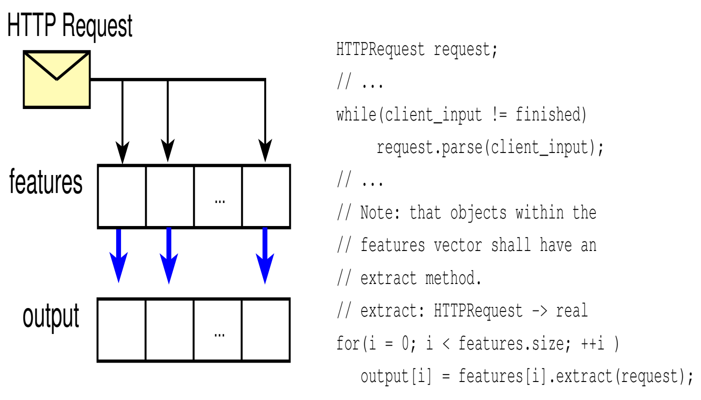


Figure 4.16: Control-flow diagram for the CIDF Thread Events box.



```
HTTPRequest request;
// ...
while(client_input != finished)
     request.parse(client_input);
// ...
// Note: that objects within the
// features vector shall have an
// extract method.
// extract: HTTPRequest -> real
for(i = 0; i < features.size; ++i )
   output[i] = features[i].extract(request);
```

Figure 4.17: Pseudo-code demonstrating how HTTP request features are extracted.

black-list features or is originated from a forbidden IP address.

```
GET http://www.google.com/../%00cmd.exe%00\P HTTP/1.1
Host: neodimio.egmont-kol.dk
Agent: Mozilla (Debian GNU/Linux Gecko Engine 3.74-1)
Connection: keep-alive
Keep-alive: 10000
```

For example, for the HTTP request presented above, and the four FDL elements declared as:

```
header.Agent.size
header.Connection.size
header.Agent.size = 4 * body.size
request.Method.type
request.URI.ocurrences(%00)
```

when each parser engine is ran, they should return each a double value. The features extracted for the neural networks to process would is to an output data-structure, a vector. This vector, in the example request and features, contains listed from first to last: a 46.0, a 10.0, a 0.0, a 3.0 (refer to appendix E for HTTP method encodings) and a 2.0.

The control-flow within the analysis box is presented in figure 4.18. Within this diagram, a server transaction is represents the flow of data to and from the Web-Server. That is, the client request is forwarded to the server and its associated server response received.

Notice that if the client request, fires a forbidden alarm or is found to match a temporal pattern, relevant countermeasures must be carried out. In case it is the forbidden alarm that has been recognized, the countermeasure is to reply without delay a 404 status-code server response. Otherwise, the suspicious source entity that met a pattern, must be delayed. The delay procedure is depicted in figure 4.19. The type of temporal pattern defined i.e. whether it be for IP address or alarm number, determines what type what type of source entity are countermeasures taken against.

There are two types of analysis that may be deducted from figure **??**; stateless validation and stateful validation. Stateless validation involves the passing of the request features through all of the neural networks. This step hints the system as to what request might form part of a stateful attack scenario, and which will not. The accuracy with which the system is trained, depends on the generality of the training data. If the stateless analysis phase, figure **??**, raises any alarms then stateful inspection begins.

When stateful inspection begins, if the alarm set off does not belong to the forbidden ones. Then the suspicious request is forwarded to the server. Hence, a server transaction is carried out in order to have all of the relevant information on the suspicious source entity item. The sequence of steps involved during stateful is depicted in figure 4.21.

The process of pattern extraction involves three elements: the source entity container, the temporal patterns and the current request source entity item. As stated in system in the data-structures architecture, a source entity item characterizes a suspicious request by it client IP

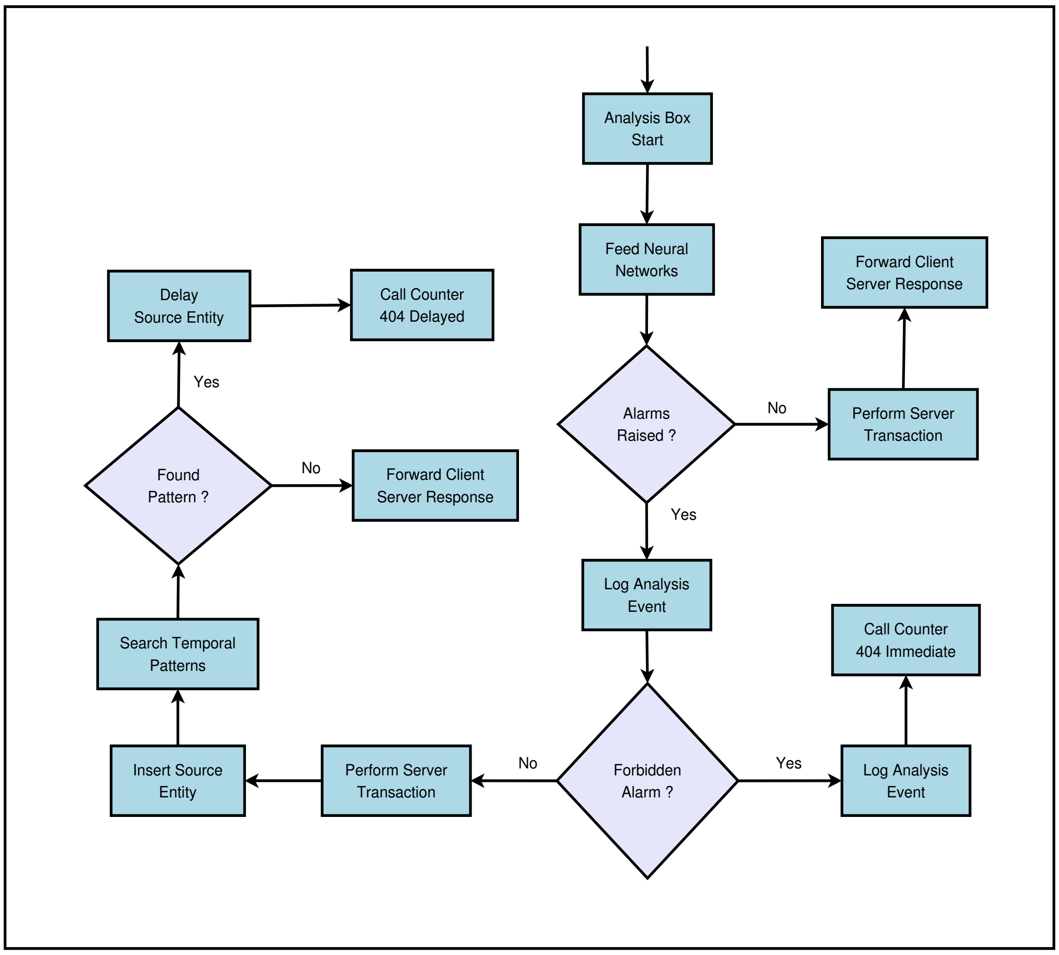


Figure 4.18: Control-flow diagram for the CIDF Thread Analysis box.

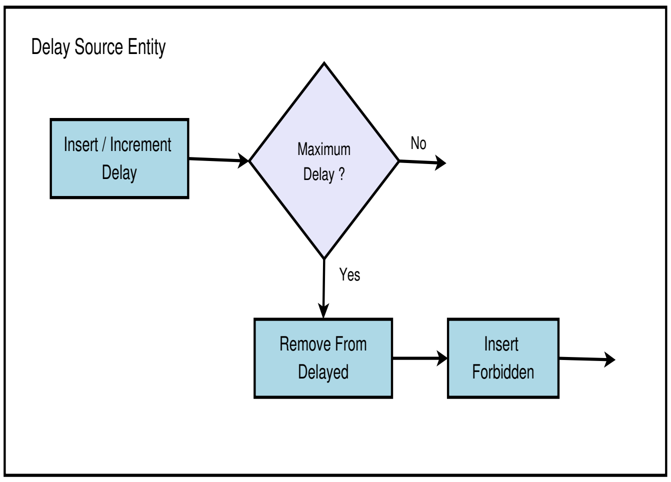


Figure 4.19: Delay Source Entity functional components internal workings.

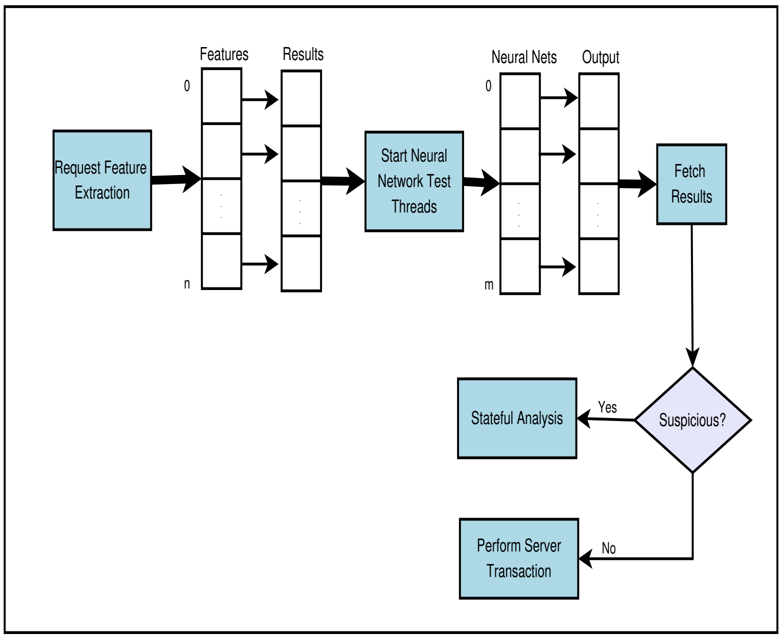

Request Feature Extraction
Features
Results
0
n
Start Neural Network Test Threads
Neural Nets
Output
0
m
Fetch Results
Suspicious?
Yes
Stateful Analysis
No
Perform Server Transaction

Figure 4.20: Component interaction during stateless validation and the neural networks.

address, the alarm it set off and the server response status-code associated with it. The parser engine for temporal patterns will search through the source entity container, for any matches that the current suspicious request may have with any of the defined patterns. The search is performed after the new source entity is inserted or an existing one is updated with its corresponding server response status-code. The steps that emphasis these events are enumerated as yellow circles in figure 4.21. Notice that steps two and three are not relevant with the source entity container. Therefore, these are present in the control-flow but not in the depiction of the data-structure.

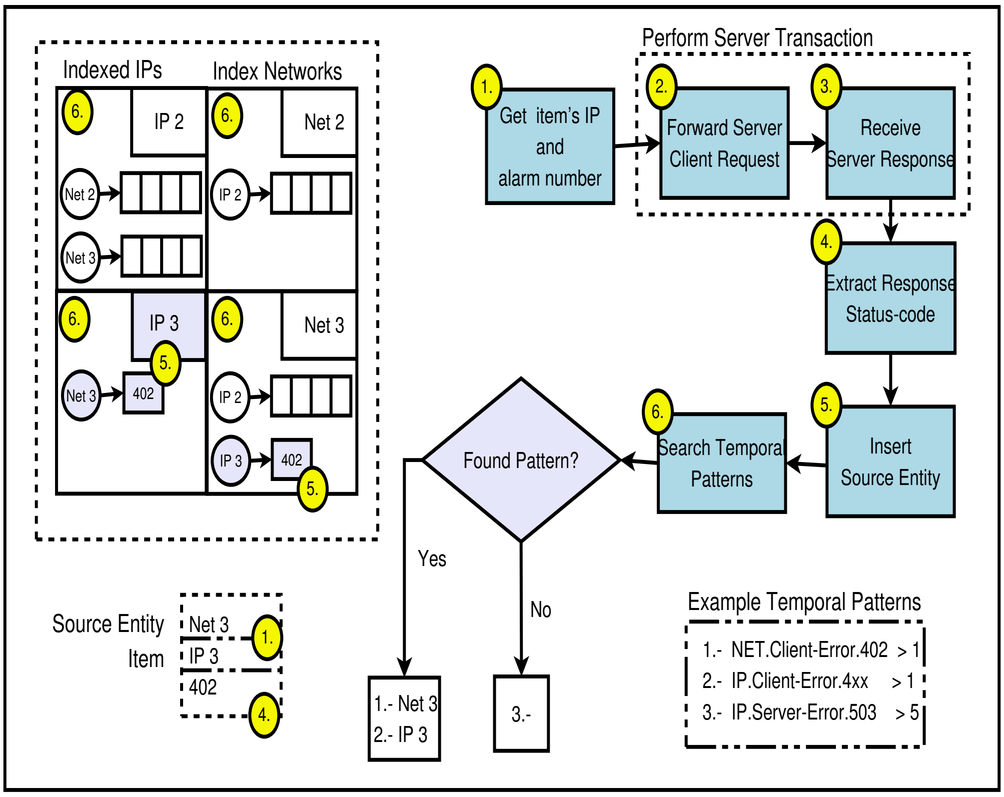

Indexed IPs
Index Networks
IP 2
Net 2
Net 3
IP 3
402
Perform Server Transaction
1.
Get item's IP and alarm number
2.
Forward Server Client Request
3.
Receive Server Response
4.
Extract Response Status-code
5.
Insert Source Entity
6.
Search Temporal Patterns
Found Pattern?
Yes
No
1.- Net 3
2.- IP 3
3.-
Source Entity Item
Example Temporal Patterns
1.- NET.Client-Error.402 > 1
2.- IP.Client-Error.4xx > 1
3.- IP.Server-Error.503 > 5

Figure 4.21: Component interaction during stateful validation and the Source Entity container.

Chapter 5

System Implementation

5.1 Overview

This chapter does not provide the source code to the project. The source code may be consulted in the attached CD. This chapter is broken up into three section: a section on how a networking environment was simulated, hierarchical diagrams on all major system classes and the complications that were encountered during the implementation. For example, how polymorphism and class inheritance were inserted into the system in order to facilitate certain tasks.

Regarding implementation of FDL or TPDL, refer to appendices E and F. These provide a listing of the implementation's public members. For a full description on the languages, refer to the attached CD-ROM. There is a source-code browsable interface that should allow a reader to easily navigate all source-code developed during this project.

The programming language used during implementation was C++. The system requires tight temporal constraints, which is why not higher level programming languages were used. For example, Java would have optimized many programming oriented issues, since there are many utilities built into the system's standard libraries. These libraries are not available for C++ at such a high level, although the language does provide the programmer with STL (Standard Template Library). Most of the basic data-structure blocks used, such as vector, hash-tables and priority queues were obtained from the STL. Also, a strong use of C++'s String class is used throughout the various system components.

C++ was chosen due to its hardware resource access timings as well as the possibility to design the system in an object-oriented manner. Object-oriented programming was chosen because of its additional data protection mechanisms that are available for data members of methods. The compiler used was GNU's g++ (GCC) v3.3.4, on a Debian GNU/Linux v3.1 platform. Additional software libraries were used during implementation, some native to the platform and others provided by third-party organizations. The source code does not compile on standard version of GNU's g++ version 2.95. This is due to some unstable enhancement features added to the compiler. These enhancement features are not available for standard versions because as they can built-up optimized code time and space-wise, the binaries built-up can also end-up in an undesirable state, foreign to the IDS's design.

Some of the additional third-party libraries used are:

- libxerces-c v2.6 - The Xerces c++ library is a set of utilities used to handle XML files. The FANNC library developed for this project makes strong use of the Xerces library. The Manager Console also makes use of the Xerces library to display in a structured manner the systems data-structure status. The programming interface to this library may be obtained at [21].
- libboost-regex v1.31 - Boost Regex is a C++ wrapper for the traditional POSIX regex C functions. Regex is a family of functions used to perform operations on regular expressions. That is, pattern matching with a given string of data. This library is strongly used by FDL in order to extract or find occurrences of patterns within the content of certain HTTP message-line sections. The programming interface to this library may be obtained at [22].
- librt - This library is provides real-time measurement utilities under POSIX systems. The set of functions provided, are accessed via the "time.h" system header file. Functions such as clock_gettime() provide a measurement accuracy within the order of nanoseconds. The real-time library functions are used by various IDS kernel components. These include, the threads used for training, testing, the CIDF threads and the connection manager. The purpose is to obtain statistical information on the systems speed for different purposes during testing. The programming interface to this library may be read in [23] and usage samples in [24].

Aside for these third party software libraries, a library implementing the FANNC algorithm as well as an HTTP Parser library were developed for this project. Detailed information on the design and programming interface are provided in appendix C and D. Both libraries were developed using C++ and depend strongly on the STL components.

All source-code may be browsed-through a cross-reference guide include in the CD annexed with this document. The path to the cross-reference guide is; "/doc/xref".

5.2 Networking Environment Simulation

The system was not developed under a real network environment. Instead, the system developed is a simulator that does not communicate neither with a web-server nor a web-client. In order to simulate a networking environment, a file called "http_traffic.cpp" includes constant variables that are globally available to all of the IDS's functional components. The sample content of the file may be seen in appendix (HTTP TRAFFIC). These variables are several vectors that simulate all of the required components in an HTTP client-server session. The type of of data contained within all of these vectors are strings of characters. The variables that are declared in this file represent:

- A set of sane HTTP client request i.e. legal requests. The variable is called sane_session_reqs.
- A set of sane HTTP server responses. Each of these components is associated with the components within the sane HTTP requests variable. The variable is called sane_session_ress.

- A set of suspicious HTTP client requests i.e. the requests contain undesired data. The variable is called suspicious_session_reqs.
- A set of server responses associated with the above suspicious requests. The variable is called suspicious_session_ress.
- A set of client IPs simulating a client soliciting a resource from a web-server. Depending on the simulation of malicious activity, sane activity or arbitrary activity, each component from this vector associates with its corresponding component in one of the requests vector. When the IDS fetches the server response, it fetches the response from the corresponding position one of the request vectors.

5.3 System Class List

5.3.1 IDS Kernel Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

5.4 System Class Documentation

5.4.1 CIDFThread Class Reference

The CIDFThread class.

```
#include <cidf_thread.hpp>
```

Public Member Functions

- **CIDFThread** (**SYSData** ∗data_structs_, unsigned int file_descr_, unsigned int www_-port_)

  Public constructor.

- ~**CIDFThread** ()

  Public destructor.

- void **run** ()

  Initialize the HTTP request's analysis.

Private Member Functions

- void **analysis_log** (const char ∗msg_)

  The analysis logs sub-system abstraction.

- bool **events_box** ()

  Perform the events extraction.

- void **analysis_box** ()

  Perform event analysis.

- void **counter_box** (unsigned int res_status_)

  Carry-out relevant countermeasures.

- int **storage_box** (std::string ipaddr_, unsigned int status_)

  Used to store a client IP.

- int **storage_box** (unsigned int alarm_n_, unsigned int status_)

  Used to store an alarm number.

- int **storage_box** (**SRCENTDat** ∗item_)

  Used to store a Source Entity.

Private Attributes

- HTTPRequest ∗ **client_req**

  Pointer to an HTTP client request.

- HTTPResponse ∗ **server_res**

  Pointer to an HTTP server response.

- **SYSData** ∗ **data_structs**

  Pointer to the IDS's Data-structures.

- std::vector< double > **features**

  Extracted request features.

- std::vector< double > **bl_features**

  Extracted black-list request features.

- std::vector< double > **alarms**

  Stateless Neural Network analysis result.

- std::vector< bool > **patterns**

  Stateful Temporal Patterns analysis result.

- unsigned int **net_descr**

  File descriptor connection to the client.

- unsigned int **www_port**

  Port that the W3-Server is listening on.

- std::string **analysis_fname**

  The analysis log file-name.

- std::vector< pthread_t > **test_threads**

  The test thread ID vector.

5.4.1.1 Detailed Description

This class implements the threads that perform stateless and stateful validation of HTTP client requests.

5.4.1.2 Constructor & Destructor Documentation

CIDFThread::CIDFThread (SYSData ∗ *data_structs_*, unsigned int *file_descr_*, unsigned int *www_port_*)
The constructor initializes the data-structure reference, gets the file descriptor for the client connection and the web-server's port. The constructor also prepares the test thread id vector

See also:
test_threads(p. 61)

CIDFThread::~CIDFThread ()
The destructor deletes all allocated memory to the HTTP request and response objects.

5.4.1.3 Member Function Documentation

void CIDFThread::analysis_box () `[private]`
This method is called after the events, features, have been successfully extracted. It starts stateful and stateless analysis. Stateless analysis makes use of the Neural Networks and stateful analysis makes use of the Suspicious Source Entity Container.

void CIDFThread::analysis_log (const char ∗ *msg_*) `[private]`
The logging sub-system will write a log message to the file specified by

Parameters:

See also:
syslog_fname

Parameters:
msg_ content of analysis log

Returns:
void

void CIDFThread::counter_box (unsigned int *res_status_*) `[private]`

This method will auto-generate a 404 reply and respond with to a client. The response is not immediate unless the client has an associated Source Entity Item in the Forbidden Source Entity Container. Either the Events or the Analysis Box can call this method.

Parameters:

res_status_ indicates the type of response

bool CIDFThread::events_box () `[private]`

This method must be the first step called while analyzing a client's HTTP request. It extracts the request's black-list and normal features.

void CIDFThread::run ()

This method is called by one of the threads launched by the Connection Manager and it services the clients request. This method performs the main analysis control-flow.

int CIDFThread::storage_box (SRCENTDat $*$ *item_*) `[private]`

This method is used to store a source entity in the Source Entity Container.

Parameters:

item_ Request's source entity.

Returns:

if a 0 the source entity was inserted successfully, anything else denotes an error.

int CIDFThread::storage_box (unsigned int *alarm_n_*, unsigned int *status_*) `[private]`

This method is used to store an alarm number, neural network index number, in the delayed source entity items Container.

Parameters:

alarm_n_ source entity alarm number.

status_ UNDEF

Returns:

if a -1 is returned an error occurred and the the insertion was not completed. If a 1 is returned then the source entity item was delayed. If a 0 is returned the source entity was removed from delayed and inserted in forbidden.

int CIDFThread::storage_box (std::string *ipaddr_*, unsigned int *status_*) `[private]`

This method is used to store a client IP in the delayed source entity items Container.

Parameters:

*ipaddr_* source entity IP address.

*status_* UNDEF

Returns:

if a -1 is returned an error occurred and the the insertion was not completed. If a 1 is returned then the source entity item was delayed. If a 0 is returned the source entity was removed from delayed and inserted in forbidden.

5.4.1.4 Member Data Documentation

std::vector<double> CIDFThread::alarms `[private]`

This vector is the result of the multi-threaded neural network testing process. The position of each neural network in the Neural Networks Container will output its result in its corresponding position here.

std::string CIDFThread::analysis_fname `[private]`

This is the name of the file where the logging sub-system shall log all analysis related activity.

std::vector<double> CIDFThread::bl_features `[private]`

This vector is the result of the black-list feature extraction process. The position of each feature in the Black-list Feature Container will output its result in its corresponding position here.

HTTPRequest* CIDFThread::client_req `[private]`

This is a pointer to the parsed HTTP request send by a client. Before this element may be used its parse method must be carried out.

SYSData* CIDFThread::data_structs `[private]`

This is a pointer to all of the IDS's data-structures i.e. the dynamic and static ones.

std::vector<double> CIDFThread::features `[private]`

This vector is the result of the feature extraction process. The position of each feature in the Feature Container will output its result in its corresponding position here.

unsigned int CIDFThread::net_descr `[private]`

This is a simulation of the file descriptor that would obtain the socket connection to the client. Instead an unsigned integer is provided and a lookup in the client_ips extern value is done.

std::vector<bool> CIDFThread::patterns `[private]`

This vector is the result of the Temporal Patterns extraction process. The position of each temporal pattern in the Temporal Patterns Container will output its result in its corresponding position here.

HTTPResponse∗ CIDFThread::server_res `[private]`

This is a pointer to the associated HTTP server response to the received server request. In case of countermeasures taken, it will be created by the IDS and sent as a decoy.

std::vector<pthread_t> CIDFThread::test_threads `[private]`

This vector is used to keep track of all the threads launched during the neural networks' testing mode. It is useful while "joining" threads.

unsigned int CIDFThread::www_port `[private]`

This is the port that the thread should use to connect to the hidden server.

The documentation for this class was generated from the following files:

- cidf_thread.hpp
- cidf_thread.cpp

5.4.2 CONNManager Class Reference

The CONNManager class.

`#include <conn_manager.hpp>`

Public Member Functions

- **CONNManager** (**SYSData** ∗data_structs_, unsigned int listen_port_, unsigned int www_port_)

  Public constructor.

- void **set_stop** (bool)

  Member method used to stop the connection manager.

- void **run** ()

  Method that initializes the the Connection Manager.

- void **reset_counter** ()

  Auxiliary network simulator function.

Private Attributes

- bool **stop**

  Enable / Disable the Connection Manager.

- **SYSData** $*$ **data_structs**

  Pointer to the IDS's Data-Structures.

- unsigned int **listen_port**

  Port to listen on for clients send connections requests.

- unsigned int **www_port**

  Port to connect to hidden web-server.

- unsigned int **counter**

  Auxiliary counter for network simulation.

5.4.2.1 Detailed Description

This class is the component that accepts client connections. Upon a client connection this component launches a **CIDFThread**(p. 55) to service the request. It does not use a thread pool, due to a DoS attack described in the system design specification.

5.4.2.2 Constructor & Destructor Documentation

CONNManager::CONNManager (SYSData $*$ ***data_structs_*, unsigned int *listen_port_*, unsigned int *www_port_*)**

The constructor initializes the listen_port and www_port components. The stop flag is initialized to false and the counter reseted to zero.

Parameters:

$*$ data_structs_ pointer to Kernel Data-Structures.

listen_port_ port where clients connect to.

www_port_ port where the hidden server is listening on.

See also:

data_structs(p. 63)

listen_port(p. 63)

www_port(p. 63)

5.4.2.3 Member Function Documentation

void CONNManager::reset_counter ()

This method is called from the MCI. It is used to re-read the http traffic from its vector containers.

void CONNManager::run ()

This method sets the Connection Manager in listening mode. It also spawns CIDFThreads when a client connection request is acknowledged.

void CONNManager::set_stop (bool)

This method indicates the connection manager that either training will begin, or the system is shutting down. The Connection Manager does not shut-down until all of its CIDFThreads are finished.

5.4.2.4 Member Data Documentation

unsigned int CONNManager::counter `[private]`

The value of this counter indicates to the CIDFThreads which component of the http traffic vector should be accessed. HTTP traffic vectors include sane requests and responses. Illegal requests and responses and the origin client IP address.

SYSData$*$ CONNManager::data_structs `[private]`

This pointer is only needed when the CIDFThreads are started. It must be passed as an argument to their constructor.

unsigned int CONNManager::listen_port `[private]`

This is the port that clients should attempt to connect to the system when it is on testing mode.

bool CONNManager::stop `[private]`

This variable is set by the user through the MCI either during system shut-down or during system training.

unsigned int CONNManager::www_port `[private]`

This is the port where the CIDFThreads will attempt to forward any valid HTTP requests.

The documentation for this class was generated from the following files:

- conn_manager.hpp
- conn_manager.cpp

5.4.3 DELAYEntities Class Reference

Delayed Source Entities Container.

```
#include <src_ent.hpp>
```

Public Member Functions

- **DELAYEntities** (double max_delay_, double incr_)

  Public constructor.

- double **get_ip_delay** (std::string ipaddr_)

  Get the IP address's delay time.

- double **get_alarm_delay** (unsigned int alarm_n_)

  Get the alarm number's delay time.

- int **delay_ipaddr** (std::string ipaddr_)

  Insert or increment the IP address's delay time.

- int **delay_alarm** (unsigned int alarm_n_)

  Insert or increment the alarm number's delay time.

- void **erase_ipaddr** (std::string ipaddr_)

  Delete IP address from Delayed Source Entity IPs.

- void **erase_alarm** (unsigned int alarm_n_)

  Delete Alarm number from Delayed Source Entity alarms.

- const char ∗ **to_string** () const

  Serialize the Delayed Source Entities to XML string.

Private Attributes

- double **max_delay**

  The user-defined maximum delay.

- double **incr**

  The user-defined delay increment.

- DELAYNetHash **delayed_alarms**

  Map containing delayed alarm numbers.

- DELAYIPHash **delayed_ips**

  Map containing delayed IP addresses.

5.4.3.1 Detailed Description

This class contains two maps of Delayed source entity items. One is indexed by Delayed IPs and the other by sounded off alarm numbers. These maps are not associative in their layout.

5.4.3.2 Constructor & Destructor Documentation

DELAYEntities::DELAYEntities (double *max_delay_*, double *incr_*)

This public constructor initializes the maximum user-defined delay time and the user-defined increment. User-defined parameters are defined in the .conf file.

Parameters:

max_delay_ maximum user-defined delay time.

incr_ user-defined increment.

5.4.3.3 Member Function Documentation

int DELAYEntities::delay_alarm (unsigned int *alarm_n_*)

Increment the alarm number's delay time. If it does not exist insert it.

Parameters:

alarm_n_ delayed alarm number.

Returns:

status -1.0 error else update successful.

int DELAYEntities::delay_ipaddr (std::string *ipaddr_*)
Increment the IP address's delay time. If it does not exist insert it.

Parameters:
ipaddr_ delayed IP address.

Returns:
status -1.0 error else update successful.

void DELAYEntities::erase_alarm (unsigned int *alarm_n_*)
Remove an entry from the Delayed Source Entity alarms vector. This activity is carried out when the alarm's delay time is greater then the maximum delay.

Parameters:
alarm_n_ Alarm number to delete.

void DELAYEntities::erase_ipaddr (std::string *ipaddr_*)
Remove an entry from the Delayed Source Entity IPs vector. This activity is carried out when an IP's delay time is greater then the maximum delay.

Parameters:
ipaddr_ IP address to delete.

double DELAYEntities::get_alarm_delay (unsigned int *alarm_n_*)
Get the alarm number's delay time.

Parameters:
alarm_n_ delayed alarm number.

Returns:
alarm number's delay time.

double DELAYEntities::get_ip_delay (std::string *ipaddr_*)
Get the IP address's delay time.

Parameters:
ipaddr_ delayed IP address.

Returns:
IP address's delay time.

const char ∗ DELAYEntities::to_string () const

This method translates the content of the Delayed Source Entities Container into an XML document.

Returns:

an XML formatted string.

5.4.3.4 Member Data Documentation

DELAYNetHash DELAYEntities::delayed_alarms `[private]`

This map contains elements of: typedef std::map<unsigned int, double> DELAYNetHash

DELAYIPHash DELAYEntities::delayed_ips `[private]`

This map contains elements of: typedef std::map<std::string, double> DELAYIPHash

double DELAYEntities::incr `[private]`

This increment is done by multiplication, i.e. current_delay := current_delay ∗ incr, hence defining an exponential behavior.

The documentation for this class was generated from the following files:

- src_ent.hpp
- src_ent.cpp

5.4.4 FORBIDDENEntities Class Reference

Forbidden Source Entities Container.

`#include <src_ent.hpp>`

Public Member Functions

- **FORBIDDENEntities** ()

  Public constructor.

- void **insert_ipaddr** (std::string ipaddr_)

  Insert a Source Entity IP address.

- void **insert_alarm** (unsigned int alarm_n_)

  Insert a Source Entity alarm number.

- bool **forbid_ipaddr** (std::string ipaddr_) const

Check if ipaddr_ is within the forbidden list.

- bool **forbid_alarm** (unsigned int alarm_n_) const

  Check if alarm_n_ is within the forbidden list.

- const char ∗ **to_string** () const

  Serialize the Forbidden Source Entities to XML string.

Private Attributes

- FORBIDDENNetHash **forbidden_alarms**

  Map containing forbidden alarm numbers.

- FORBIDDENIPHash **forbidden_ips**

  Map containing forbidden IP addresses.

5.4.4.1 Detailed Description

This class contains two maps of Forbidden source entity items. One is indexed by Forbidden IPs and the other by sounded off alarm numbers. These maps are not associative in their layout.

5.4.4.2 Member Function Documentation

bool FORBIDDENEntities::forbid_alarm (unsigned int *alarm_n_*) const

Check if alarm_n_ is within the forbidden list.

Parameters:

alarm_n_ Source Entity alarm number.

Returns:

true if

Parameters:

is forbidden false otherwise

bool FORBIDDENEntities::forbid_ipaddr (std::string *ipaddr_*) const

Check if ipaddr_ is within the forbidden list.

Parameters:

*ipaddr_* Source Entity IP address.

Returns:

true if

Parameters:

is forbidden false otherwise

void FORBIDDENEntities::insert_alarm (unsigned int *alarm_n_*)

Insert a Source Entity alarm number.

Parameters:

*alarm_n_* Source Entity alarm number.

void FORBIDDENEntities::insert_ipaddr (std::string *ipaddr_*)

Insert a Source Entity IP address.

Parameters:

*ipaddr_* Source Entity IP address.

const char ∗ FORBIDDENEntities::to_string () const

This method translates the content of the Forbidden Source Entities Container into an XML document.

Returns:

an XML formatted string.

5.4.4.3 Member Data Documentation

FORBIDDENNetHash FORBIDDENEntities::forbidden_alarms `[private]`

This map contains elements of: typedef std::map<unsigned int, int> FORBIDDENetHash, the key contents are the number of hits that have been registered for a forbidden source entity alarm number.

FORBIDDENIPHash FORBIDDENEntities::forbidden_ips `[private]`

This map contains elements of: typedef std::map<std::string, double> FORBIDDENIPHash, the key contents are the number of hits that have been registered for a forbidden source entity IP address.

The documentation for this class was generated from the following files:

- src_ent.hpp
- src_ent.cpp

5.4.5 IDSKernel Class Reference

The IDSKernel class.

`#include <idskernel.hpp>`

Public Member Functions

- **IDSKernel** ()

  Public constructor.

- ~**IDSKernel** ()

  Public destructor.

- void **init_mode** ()

  Switch the kernel to initialization mode.

- void **train_mode** ()

  Switch the kernel to training mode.

- void **test_mode** ()

  Switch the kernel to testing mode.

- const char ∗ **getmci_nets** (const char ∗net_n_)

  MCI method - Extract the state of Neural Net n.

- const char ∗ **getmci_delayed** ()

  MCI method - Extract the state of the Delayed Entities Bin.

- const char ∗ **getmci_srcents** ()

  MCI method - Extract the state of the Source Entity Bin.

- const char ∗ **getmci_forbidden** ()

  MCI method - Extract the state of the Forbidden Entities Bin.

- void **setmci_exit** ()

  MCI method - Stop the IDS.

- void **setmci_train** ()

  MCI method - Prepare the IDS for training.

Private Member Functions

- void **system_log** (const char ∗msg_)

  The system logs sub-system abstraction.

Private Attributes

- **CONNManager** ∗ **conn_manager**

  Pointer to the IDS's Connection Manager.

- **SYSData** ∗ **data_structs**

  Pointer to the IDS's Data-Structures.

- std::string **syslog_fname**

  The system log file-name.

- std::vector< pthread_t > **train_threads**

  The train thread ID vector.

5.4.5.1 Detailed Description

This class is the back-bone of the W3-IDS. It binds the data-structures with the Connection manager. It contains method used in the MCI.

See also:
    Stateless

5.4.5.2 Constructor & Destructor Documentation

IDSKernel::IDSKernel ()

The IDS's constructor initializes the system's data-structure along with the connection manager. It also prepares the train_threads for usage.

See also:

train_threads(p. 74)

IDSKernel::~IDSKernel ()

The IDS's destructor releases the memory used by the Connection Manager and the Datastructures.

See also:

conn_manager(p. 74)

data_structs(p. 74)

5.4.5.3 Member Function Documentation

const char $*$ IDSKernel::getmci_delayed ()

This method forms part of the MCI, it extracts the state of the Delayed Entities Container.

Returns:

the state in an XML formatted string.

const char $*$ IDSKernel::getmci_forbidden ()

This method forms part of the MCI, it extracts the state of the kernel's Forbidden Entities Container.

Returns:

the state in an XML formatted string.

const char $*$ IDSKernel::getmci_nets (const char $*$ *net_n_*)

This method forms part of the MCI, it extracts the state of neural network net_n_ from the neural network container.

Parameters:

net_n_ the selected neural network.

Returns:

the state in an XML formatted string.

const char ∗ IDSKernel::getmci_srcents ()

This method forms part of the MCI, it extracts the state of the kernel's Source Entity Container.

Returns:

the state in an XML formatted string.

void IDSKernel::init_mode ()

Initializes all of the components within the data_structs instance. These vectors are the features, the black-list, the neural networks and the temporal patterns.

See also:

data_structs(p. 74)

void IDSKernel::setmci_exit ()

This method forms part of the MCI, it stops the Connection Manager. This method must be called before destroying the kernel.

See also:

conn_manager(p. 74)

void IDSKernel::setmci_train ()

This method forms part of the MCI, it stops the Connection Manager and starts the training process. When training is finished, the Connection Manager is re-enabled.

See also:

conn_manager(p. 74)

void IDSKernel::system_log (const char ∗ *msg_*) `[private]`

The logging sub-system will write a log message to the file specified by

Parameters:

See also:

syslog_fname(p. 74)

Parameters:

msg_ content of system log

Returns:

void

void IDSKernel::test_mode ()

Test mode means that the Connection Manager is enabled and listening for client connection requests.

See also:

conn_manager(p. 74)

void IDSKernel::train_mode ()

Start the multi-threaded training process. During train_mode real-time timings are carried out. The results are the temporal cost (ns) from the training of each thread and the average training time for all threads.

5.4.5.4 Member Data Documentation

CONNManager∗ IDSKernel::conn_manager `[private]`

The Connection Manager is referenced through a pointer. The Connection Manager is created with the IDSKernel public constructor. And it is deleted with the destructor.

SYSData∗ IDSKernel::data_structs `[private]`

The Data-structures are referenced through this pointer. These are created when the kernel is started through **init_mode()**(p. 73). All dynamic and static vectors reside within this object.

std::string IDSKernel::syslog_fname `[private]`

This is the name of the file where the logging sub-system shall log all system related activity.

std::vector<pthread_t> IDSKernel::train_threads `[private]`

This vector is used to keep track of all the threads launched during **train_mode()**(p. 74). It is useful while "joining" threads.

The documentation for this class was generated from the following files:

- idskernel.hpp
- idskernel.cpp

5.4.6 IPIndex Class Reference

Suspicious Source entity IP item.

`#include <src_ent.hpp>`

Public Member Functions

- **IPIndex** ()

  Public constructor.

Public Attributes

- std::string **ip_addr**

  Item's IP address.

- std::map< int, std::vector< std::string > ∗ > **s_ents**

  The alarms that the IP has sounded off.

5.4.6.1 Detailed Description

This class represents an item of a Suspicious Source Entity. A suspicious source Entity is characterized by a suspicious IP and an alarm number.

5.4.6.2 Member Data Documentation

std::string IPIndex::ip_addr

This Suspicious Source Entity Item represents an IP address that has set n number of alarms off.

std::map<int, std::vector<std::string>∗ > IPIndex::s_ents

These are the alarms that are associated to the IP address. Each alarm is a unique key in the associative map. The content of each key is a vector containing associated server responses.

The documentation for this class was generated from the following files:

- src_ent.hpp
- src_ent.cpp

5.4.7 NETIndex Class Reference

Suspicious Source entity Alarm number item.

`#include <src_ent.hpp>`

Public Member Functions

- **NETIndex** ()

  Public constructor.

Public Attributes

- int **net_id**

  Item's alarm number.

- std::map< std::string, std::vector< std::string > * > **s_ents**

  The IPs that have sounded the alarm off.

5.4.7.1 Detailed Description

This class represents an item of a Suspicious Source Entity. A suspicious source Entity is characterized by a suspicious IP and an alarm number.

5.4.7.2 Member Data Documentation

int NETIndex::net_id

This Suspicious Source Entity Item represents an Alarm number that has been set off by n number of IP addresses.

std::map<std::string, std::vector<std::string>* > NETIndex::s_ents

These are the IPs that are associated to an alarm number. Each IP is a unique key in the associative map. The content of each key is a vector containing associated server responses.

The documentation for this class was generated from the following files:

- src_ent.hpp
- src_ent.cpp

5.4.8 SRCENTDat Class Reference

The Suspicious Source Entity class.

`#include <src_ent.hpp>`

Public Member Functions

- **SRCENTDat** (const char *client_ip_, const char *net_num_)

  Public constructor.

- const char * **get_svr_res** () const

  Get the server response status-code.

- const char ∗ **get_client_ip** () const

  Get the client's IP address.

- int **get_net_num** () const

  Get the alarm number sounded off.

- void **set_svr_res** (const char ∗)

  Set the server response status-code.

Private Attributes

- std::string **svr_res**

  The associate server response status-code.

- std::string **client_ip**

  The client's IP address.

- int **net_num**

  The alarm number sounded off.

5.4.8.1 Detailed Description

This class represents the contents of a suspicious source entity. It contains the client IP address, the alarm number sounded off and the associated server response status-code to a suspicious HTTP request.

5.4.8.2 Constructor & Destructor Documentation

SRCENTDat::SRCENTDat (const char ∗ *client_ip_*, const char ∗ *net_num_*)

This public constructor is used to set the client IP address and alarm number of this suspicious source entity.

Parameters:

client_ip_ the client's IP address.

net_num_ the alarm number sounded off.

5.4.8.3 Member Function Documentation

const char∗ SRCENTDat::get_client_ip () const `[inline]`

This is a constant in-line method for retrieving the client's IP address that sent a suspicious HTTP request.

Returns:

the client's IP address.

int SRCENTDat::get_net_num () const `[inline]`

This is a constant in-line method for retrieving the alarm number that a suspicious HTTP request has sounded off.

Returns:

the alarm number sounded off.

const char∗ SRCENTDat::get_svr_res () const `[inline]`

This is a constant in-line method for retrieving the server's response status-code from a suspicious HTTP request.

Returns:

the associated server's response status-code

void SRCENTDat::set_svr_res (const char ∗)

After a suspicious request has been acknowledged as non-critical it is forwarded to the server. This method is used to set the associated server's response status-code to a suspicious request.

The documentation for this class was generated from the following files:

- src_ent.hpp
- src_ent.cpp

5.4.9 SRCEntities Class Reference

Suspicious Source Entity Container.

`#include <src_ent.hpp>`

Public Member Functions

- **SRCEntities** ()

  Public constructor.

- ∼**SRCEntities** ()

  Public destructor.

- void **insert_ip** (**SRCENTDat** item_)

  Perform a unique IP address insertion.

- void **insert_net** (**SRCENTDat** item_)

  Perform a unique Alarm number insertion.

- const char ∗ **to_string** () const

  Serialize the Source Entity Container to XML string.

Private Member Functions

- void **insert_ip_index** (**SRCENTDat**)

  Insert and Source Entity IP Item.

- void **insert_net_index** (**SRCENTDat**)

  Insert and Source Entity Alarm number.

Private Attributes

- std::vector< **IPIndex** ∗ > **ips**

  The Suspicious Source Entity IPs.

- std::vector< **NETIndex** ∗ > **nets**

  Ths suspicious Source Entity Alarm number.

Friends

- class **TMPVar**

  The Temporal Patterns variable.

5.4.9.1 Detailed Description

This class contains two vectors of suspicious source entity items. One is indexed by suspicious IPs and the other by sounded off alarm numbers. The data in both vectors has an associative layout. The data in one vector is contained in the other and vice-versa. But the data is interpreted in a different way.

5.4.9.2 Constructor & Destructor Documentation

SRCEntities::~SRCEntities ()
This method is in charge of cleaning out the IPs and alarm numbers vectors.

5.4.9.3 Member Function Documentation

void SRCEntities::insert_ip (SRCENTDat *item_*)
This method inserts an IP address into the its vector. It will search for the IP address within

Parameters:

If it is found it will insert the alarm sounded off. If no alarm number is found a new one with its associated server's response status-code is, or the associated server's response status-code is appended to the existing entry.

item_ the content of a Suspicious Source Entity.

void SRCEntities::insert_ip_index (SRCENTDat) `[private]`
This method directly pushes-back a Source Entity IP Item in its vector.

void SRCEntities::insert_net (SRCENTDat *item_*)
This method inserts an alarm number into the its vector. It will search for the alarm number within

Parameters:

If it is found it will insert the IP that sounded off the alarm. If no IP is found a new one with its associated server's response status-code is, or the associated server's response status-code is appended to the existing entry.

item_ the content of a Suspicious Source Entity.

void SRCEntities::insert_net_index (SRCENTDat) `[private]`
This method directly pushes-back a Source Entity alarm number in its vector.

const char $*$ SRCEntities::to_string () const
This method translates the content of the Source Entity Container into an XML document.

Returns:

an XML formatted string.

5.4.9.4 Friends And Related Function Documentation

friend class TMPVar `[friend]`

The Temporal Patterns variable must perform lookups in both the Source Entity Container.

5.4.9.5 Member Data Documentation

std::vector<IPIndex ∗> SRCEntities::ips `[private]`

This vector contains all IPs that have sounded n number of alarms off (n > 0).

std::vector<NETIndex ∗> SRCEntities::nets `[private]`

This vector contains all of the alarm numbers that have been sounded off by n number of IP addresses (n > 0).

The documentation for this class was generated from the following files:

- src_ent.hpp
- src_ent.cpp

5.4.10 SYSData Class Reference

The SYSData class.

`#include <sys_data.hpp>`

Public Member Functions

- **SYSData** ()

  Public Constructor.

- ~**SYSData** ()

  Public Destructor.

Protected Attributes

- std::vector< ExpLangObject ∗ > **features**

  The Features Container.

- std::vector< ExpLangObject ∗ > **blacklist**

  The Black-list Features Container.

- std::vector< TMPPattern ∗ > **tmp_patterns**

  The Temporal Patterns Container.

- std::vector< FANNCNetwork ∗ > **neural_nets**

  The Neural Network Container.

- **SRCEntities** ∗ **src_ents**

  The Suspicious Source Entity Container.

- **DELAYEntities** ∗ **delayed_ents**

  The Delayed Source Entities Container.

- **FORBIDDENEntities** ∗ **forbidden_ents**

  The Forbidden Source Entity Container.

Friends

- class **CIDFThread**

  The **CIDFThread**(p. 55) *has access kernel Data-Structure.*

- class **IDSKernel**

  The **IDSKernel**(p. 70) *has access its Data-Structure.*

5.4.10.1 Detailed Description

This class holds all kernel Data-Structures. Including both dynamic and static types.

5.4.10.2 Constructor & Destructor Documentation

SYSData::~SYSData ()

The public destructor is in charge of freeing all memory occupied by the dynamic Data-Structures.

5.4.10.3 Friends And Related Function Documentation

friend class CIDFThread `[friend]`

CIDFThread(p. 55) class is allowed to directly manipulate the IDSs Data-Structures.

friend class IDSKernel `[friend]`

IDSKernel(p. 70) class is allowed to directly manipulate the Data-Structures.

5.4.10.4 Member Data Documentation

std::vector<ExpLangObject ∗> SYSData::blacklist `[protected]`

This container is static. It holds all of the user-declared black-list features. Each element in this vector has its own parser engine.

DELAYEntities∗ SYSData::delayed_ents `[protected]`

This container is dynamic. It holds all source entities characteristic items that have been delayed. Characteristic items may be an IP address or an alarm number.

std::vector<ExpLangObject ∗> SYSData::features `[protected]`

This container is static. It holds all of the user-declared features. Each element in this vector has its own parser engine.

FORBIDDENEntities∗ SYSData::forbidden_ents `[protected]`

This container is dynamic. It holds all source entity characteristic items that have reached their maximum delay time.

std::vector<FANNCNetwork ∗> SYSData::neural_nets `[protected]`

This container is static. It holds all of the stateless validation neural networks. Each neural network in this vector is also interpreted as an alarm.

SRCEntities∗ SYSData::src_ents `[protected]`

This container is dynamic. It holds all of the source entities that have fired a stateless alarm. Alarms are fired due to suspicious content within the HTTP request.

std::vector<TMPPattern ∗> SYSData::tmp_patterns `[protected]`

This container is static. It holds all of the user-declared temporal patterns. Each element in this vector has its own parser engine.

The documentation for this class was generated from the following files:

- sys_data.hpp
- sys_data.cpp

5.5 Complications

During the implementation phase there were two main complication issues. These had to be solved using mechanisms provided by the programming language. The first implementation issue was working out a simple yet robust programming abstraction for both of the recursive languages. That is, the languages used to extract user-defined features (FDL) and temporal patterns (TPDL). The second issue was making a wrapper around the kernel to enable user interaction. This refers to the MC and the kernel's MCI. With another programming language, such as C, these implementation issues would have been complex to solve. C is mentioned because it offers about the same hardware access time as C++. This was the main reason why Java was not an option for this project.

During the implementation of FDL and TPDL, a strong use of C++'s polymorphic mechanism had to be used. Notice that FDL and TPDL share a common design, and are nearly identical in their implementation. There are three main differences:

- The variable look-up table that each uses is different. FDL uses a parsed request object as a look-up table while TPDL uses the Suspicious Source Entity Container.
- During the parsing of user-defined features and temporal patterns, the variable declarations do not share the same syntax.
- TPDL does can only interpret boolean expressions unlike FDL that can, e.g. the size of the HTTP header-field content.

Polymorphism allows for the recurrent traversal of data among the description language's expressions. In other words, the user may declare nested user-defined features and temporal patterns. For example,

- (ALARM.Client-Error.4xx > 400) and (IP.Redirection.3xx > 5)

The other issue, was implementing the CM and the kernel's CMI. Through object inheritance offered by the languages facilities, this task was reduced to a few lines of code. An CM class is derived from the kernel. Therefore, the CM is itself a kernel with added-on features. The way that the CMI is defined is by privatizing all data-member within the kernel, except those available to the CMI. These are defined as "protected" within the kernel, therefore only derived kernel objects may access them. No data nor method members should be left public within the kernel. With a well defined CMI it is easy to derive different types of user interfaces. Not necessarily a console interface but a GUI could also be used.

Chapter 6

System Testing

6.1 Overview

The description of the IDS's testing is divided into three parts: What type of equipment and auxiliary programs were used, how relevant data was gathered and the system's training and testing times. Notice that these tests are subject to change from the delivery of this document to the system's presentation day. The system is still in a prototype stage and it is still constantly being updated. The prototype delivered with this along with this document is the latest stable version.

Notice that all testing data used; temporal as well as HTTP dissector results, are contained with the CD annexed to this document.

6.2 Testing Environment

Notice that all tests were performed in an isolated environment. That is, there was never access to the test servers nor the IDS from a DMZ. All tests were performed from the local-host computer or using an auxiliary lap-top computer within the IMM firewall's perimeter.

6.2.1 Equipment

All test were performed on two x86-based systems. The first running on an AMD Athlon(™) XP 2000+ at 1.6 GHz processor. The available memory was 256MB SDRAM. The OS running the machine was a Debian GNU/Linux 3.1. Some data characterization requests were launched from a second system. This was a Toshiba Satellite 2430-301 lap-top computer. The lap-top has is an x86-based system running on a Intel(™) Pentium 4 at 2.66 GHz processor. The amount of memory available on this computer where 512 DDR-SDRAM. The OS was again a Debian GNU/Linux 3.1 and Windows XP Professional Edition.

6.2.2 Auxiliary Applications

There were various different types of auxiliary applications used to perform the tests. These are:

- An Apache Web-Server version 1.3.33 on system 1. The web-server was used as a sink for all captured test HTTP requests. These include sane HTTP requests from stress tools and normal web-browsers to suspicious activity emulated with Nessus.

- Etherreal version 0.10.6 on system 1. This tool is a Ethernet packet sniffer and was used to intercept all tested HTTP traffic. This traffic was then organized as application-level data (strings) and placed inside the prototype. The HTTP traffic, as mentioned before, resides globally accessible vectors within the IDS, enabling network simulation.

- Nessus version 2.2.0 on system 1. This tool was used to mimic suspicious and critical activity against the test web-server. It can mimic XSS attacks, various DoS attacks, BoF attacks and various forms of injection attacks. These injection attacks include SQL database tampering.

- MyPHPMoney version 1.3RC3 on system 1. This tool makes use of the MySQL server through the PHP scripting sub-system of Apache. It was used to launch and see if any injection attacks could be performed and what information was obtained from the analyzed traffic.

- MySQL Server version 4.0.23 on system 1. This server was used as a requirement of MyPHPMoney and to study the effects of automatic injection attacks from Nessus. These attacks were performed through the PHP scripting sub-system.

- Mozilla Firefox version 1.0 on system 2. This tool was used to extract properties of normal web-traffic. It was fundamental in differentiating those features from normal and suspicious requests.

- Opera version 7.54u1 on system 2. This tool was used to extract properties of normal web-traffic. It was fundamental in differentiating those features from normal and suspicious requests.

- Internet Explorer version 6.0 on system 2. This tool was used to extract properties of normal web-traffic. It was fundamental in differentiating those features from normal and suspicious requests.

- Hammerhead version 2.1.3 on system 1. This is a Web-Server stress tool and was used to differentiate relevant features from normal and suspicious requests. Once the prototype works on-line, this tool should be used to stress test how many requests it is capable of handling per second.

6.3 Data Gathering

All data gathered was obtained through the use of an Ethernet packet sniffer. There were automatic attacks launched through Nessus at the Web-Server. These differed in the sort of attack, for example, DoS attacks , XSS attacks, SQL Injection attacks and BoF attacks . Also data-gathering intrusions where carried out, such as techniques used by port-scanners , directory traversals and user-name account guessing techniques. Not only were malicious requests gathered, also normal HTTP traffic from ordinary Web-Browsers was sniffed. Also the normal interaction of a user with the SQL Database through Apache's PHP scripting sub-system. In order to point-out any relevant characteristics of suspicious and anomal behavior.

An auxiliary tool was used, which makes use of the HTTP Parser Library developed for this project (appendix D). The utilities usage is

```
dishttp: HTTP traffic statistical analyzer, version 0.1
  Usage: dishttp file ss1 ss2 ss3
     file   File That contains
            HTTP requests.
      ss1   Minimum segment size
            within header content.
      ss2   Minimum segment size
            within URL content.
      ss3   Minimum segment size
            within body content.
```

A sample file that contains simulated HTTP traffic can be seen in the source file "http_traffic.cpp". This file may be accessed in the CD that is annexed to this document. It resides within the directory path: "/src/W3ids/". The most relevant results obtained are displayed in tables G.2 and 6.2. Notice that for anomel traffic, there are two types of Client-Server interaction captures. A normal browsing mode and another one with the PHP scripting subsystem.

Browser	GET Usage	POST Usage	URL Range
Firefox	100%		[1..56]
Firefox PHP	90%	10%	[21..38]
Opera	100%		[1..16]
Opera PHP	90%	10%	[12..45]
IE	100%		[1..7]
IE PHP	90%	10%	[11..38]
Mozilla	100%		[1..16]
Mozilla PHP	90%	10%	[12..38]

Table 6.1: Relevant features extracted from anomal HTTP activity.

During the dissection of HTTP request in DoS attacks, the HTTP version field varied. From distinct versions to random strings. Refer to appendix G the full content of data gathering, and a detailed explanation.

Attack	GET Usage	POST Usage	OPTIONS Usage	Other method	URL Range
XSS	76%	24%			[10..144]
PHP-SQL	100%				[11..113]
Guess	100%				[3..22]
DoS	55%	5%	10%	5%	[1..9]

Table 6.2: Relevant features extracted from suspicious HTTP activity.

6.3.1 Detection Efficiency

The system's detection efficiency is hard to measure. Specially because most of the suspicious activity was performed under a control environment. Another disadvantage was that it was not easy to come across old software systems with vulnerabilities that had not been patched. The system was capable of detecting from the simulations most of the attack attempts by Nessus. Although most of the attacks were performed in one-pass mode. That is, it was only one HTTP sent and not a sequence of them. What did turn out convincing, were the user-defined temporal patterns. Although there was no auxiliary tool to analyze temporal patterns, from empirical criteria, the system was capable of recognizing several directory traversal, and user-name account guessing intrusions from the correlation of client IP, alarms sounded off and server responses.

Another problem while measuring the detection efficiency was that most user-defined features were defined by heuristics. Not through the data extracted from the HTTP dissection tool. The data that was obtained from Nessus was too general, to not be searched by regular expression matching. For example, in specific patterns within the URL field like "<SCRIPT".

6.4 System Timing

There are several timing parameters to considering while measuring the temporal efficiency of the system. These are broken up into three groups of data: the training time required by the system, the testing time required by the system and the overall time required to process a client request. All of the timings bellow can be stipulated that the server response time and network data-transfer rates are constant. Therefore, in each case where temporal parameters are obtained the following expression must be kept in mind:

$$T(n) = n + C_1 + C_2$$

Where n is the total time that the IDS used while processing the request. C_1 is the constant time that a server has in completing a Server Transaction. From chapter 4, a server transaction implies, a request being forwarded to the server and the server sending back its response. C_2 is the constant time that the network data transfer-rate adds up. All system timing parameters are given in nanoseconds (ns).

6.4.1 Training Time

There is an obvious established relation with the amount of training data fed into the networks and the time all networks require to be trained. In figure 6.1 this lineal relations is shown.

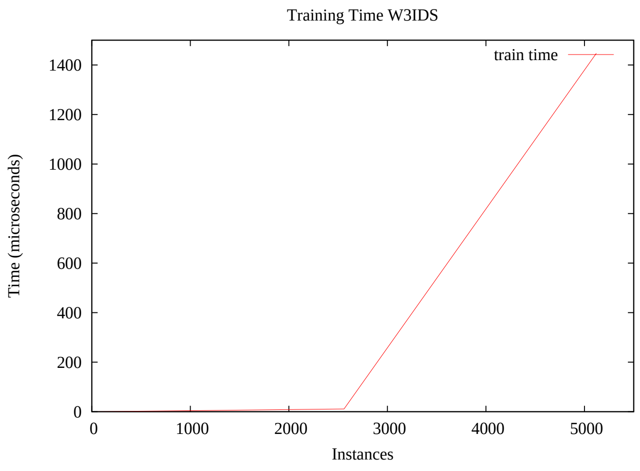


Figure 6.1: Main system data-structures components.

There is a small errata in figure 6.1, time units are presented as microseconds but are milliseconds.

6.4.2 Testing Time

The testing time remained constant for the neural networks. The amount of Internal Input Classification units in the hidden layer of the neural network had no impact. There was an average testing time of:

$$4,289\ \mu s$$

With a minimum time of $3,6\ \mu s$ and a maximum of $9,8\ \mu s$. On seldom occasions there were testing peaks of $195,6\ \mu s$. Testing time peaks could be due to system overload during the process of testing.

6.4.3 HTTP Request Service Time

The overall system request service time i.e. from the time that a client sends the HTTP request to the time it gets a response needs to be broken down. There are two main cases; the case when a request is considered sane, and undergoes no further analysis other than stateless analysis. In this case the system timing was of,

$$300\ \mu s$$

while during the works case scenario i.e. when the system must undergo stateless as well as stateful analysis. The timing averaged out to,

$$972, 25\,\mu s$$

Chapter 7

Further Developments

7.1 Overview

In this chapter a series of improvements that the system should be subject are given. These go from major system architecture changes to some minor fix-ups. Most of the developments mentioned here are services that are offered by current market products. For any IDS designed for today's Web-Application domain, most of these add-ons are fundamental system requirements.

7.2 Maximum Delay DoS Fix

In the DoS attack mentioned in Chapter 4, held against the Connection Manager. The problem is not whether the system uses a thread pool or not, to service client requests. The problem is within the design of the modified CIDF proposal.

The Countermeasures box abstraction should not be a function called from within the events and analysis box. The countermeasures box should be implemented as an independent thread. The interaction between the events, analysis box and countermeasures box should be a buffered producer, consumer semaphore protected approach.

The fix-up proposes that each time a client needs to receive a delayed decoy response, these be inserted into a buffer by the analysis or events boxes. A priority queue should be used as the buffer. Having the ordering element of the queue as the shortest delay time at first position and the largest delay time at last. The data contained in this element inserted in the shared-buffer must also have information relevant to the clients socket descriptor. In order to allow the Countermeasures box to take full responsibility of the response that the client will get.

This fix-up does not lock-up the CIDF Threads when a countermeasure must be taken against a client request. Therefore, the CIDF Threads are free to process more client requests received by the Connection Manager.

7.3 On-line Training

On-line training refers to the system not only being capable of performing training during system start-up. On-line training should be carried out while the system is up and running.

This proposal would make the stateless engine a very powerful utility if retraining from scratch was not needed. But as soon as new instances of request features were stumbled across these could be added into the neural network. This process could be automatic, although the Neural Network library developed herein would need to be extended to be able to make those kind of decisions.

What is implied here is, that the neural network would have to be trained to categorize an event as a certain class of attack, and some features that categorize the event as not an attack. Therefore, when data could not be fitted into either one of the two sets, a decision would have to be made. If the neural networks were assisted by a network administrator, he could simply decide which new instances of features were added to which nets.

Assisted on-line training could be performed from within the local-host computer or a remote computer. Under a distributed network scheme, remote access would be convenient for the system administrator. The local-host IDS management can be performed through the use of UNIX domain sockets. For security reasons remote access to the IDS would have to then, support SSL connections.

7.4 Hand-off Scheme

The Hand-off scheme is used for heavily loaded Web-Servers. Under this context, the load of a Web-Server is measured by client requests or hits per second (hits/s). Some examples of a heavily loaded Web-Servers are: the Google Web-Servers, the CNN Web-Servers, the Amazon.com Web-Servers and MSN's Hotmail Web-Servers. These Web-Servers receive loads in the order of thousands of hits per second. A single Web-Server does not have the capacity to handle such loads, so several of them are used together. A main Web-Server is placed to receive all client requests, but it does not process them. The client requests get forwarded to slave Web-Servers. These may be organized by functionality i.e. SQL oriented, or XML content pages. Or they may be oriented by the part of the site that is being visited i.e. the Web-site is distributed among several machines.

How the proposed Application-level IDS could be incorporated into this scheme is a mere educated guess. Since it has not been either subject to such Web-request loads nor have the hardware resources, allowed a hand-off scheme trial. The proposal is that an instance of each application-level IDS be placed in-front of each slave Web-Server. The IDS's would have to intercommunicate adding a data synchronization temporal impact factor on the performance of its analysis speed.

The other possibility is by having an ideal IDS in-front of the master Web-Server that could service up to 30.000[*] hits/s in real-time. Nearly impossible, since stateful inspection adds a server response temporal factor which is external to the IDS temporal timing constraints. Server

[*]The estimate is obtained from the analysis speed that Intelliwall offers, without the stateful correlation of server responses.

responses are not a constant factor either, these vary depending on the load that the machine running the Web-Server is subject to. Therefore, the only valid scheme proposal is in front of each slave Web-Server.

7.5 SSL HTTP Traffic

For most Web-Applications that involve the handling of user data, the encrypted version of the HTTP protocol is used i.e. HTTPS. Which means that any intermediate point between the client and Web-Server can receive the information, but just won't make much sense out of it. Therefore, the IDS would require a Web-Server certificate upload authority. This would allow the IDS to identify itself to the clients as the server. Forwarding then to the Web-Server either decrypted or encrypted information received from the clients.

7.6 Hash-table Over Vector

There is a minor efficiency problem in the design of the Source Entities Container. The container is characterized by two vectors; one holding all of the information relevant to indexed IP address and the other to alarms that have been sounded off. Within each vector component, there is a reciprocal Source Entity Item. This means that in the indexed IPs, each IP has a set of alarms that has set off and vice-versa. In the case of the indexed IPs, each alarm is a key within an associative map. The key returns a vector which contains all server responses that the IP address, having set off that alarm have received. Instead of a vector for storing server responses, another associative map should have been used. That is, a 2-dimensional associative map and the following cases analysis, provides an elemental mathematical proof on why.

During the case analysis vector elements are considered as constant strings of four characters e.g. "404
0". In the associative map, the key is also a constant string of four characters, and the key content is a short integer i.e. an integer composed of four bytes. The key content represents the number of time that the key has been received as a server response Status-Code. The associative map's keys are unique, they may not repeat. Repeated elements are represented by r;

7.6.1 First Case

From figure 7.1, the worst case scenario is modelled for the hash-table. There have been n Status-Codes received , up until this time frame. All of the received Status-Codes are different. Therefore, spacial-wise the associative map occupies twice as much space as the vector.

The vector under this case occupies $4 bytes * n$, while the associative map occupies $2 * 4 bytes * n$. Notice that the number of repeated elements in this case is n, therefore $r = n$.

7.6.2 Second Case

From figure 7.2, the best case scenario is modelled for the hash-table. There have been n Status-Codes received, up until this time frame. All of the received Status-Codes are the same. There-

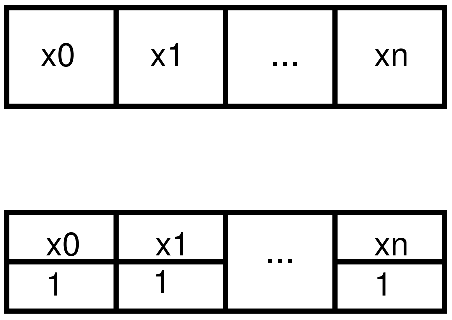


Figure 7.1: Vector VS. Hash, best case for vector data-structure.

fore, spacial-wise the associative map occupies $(1/n)+4bytes$ the size of the vector. The amount of repeated elements under this case is 1, therefore $r = 1$.

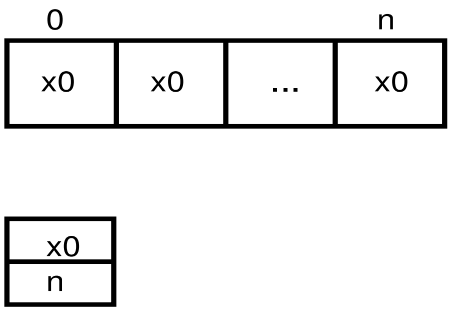


Figure 7.2: Vector VS. Hash, best case for the hash-table data-structure.

The vector in this case still occupies $4bytes*n$ bytes, while the associative map only occupies $2bytes$.

7.6.3 Conclusion

Temporal-wise it is obvious that the associative map is more efficient since its lookup times are constant, while the vectors look-up times depend on its size i.e. $O(T) = n$.

Spacial-wise the following relation is established: as soon as there are more than elements than twice the amount of repeated objects the hash-table is more efficient.

$$n > 2r \qquad \wedge \qquad n \geq r$$

From empirical results obtained during the evaluation of experiments, the number of server response Status-Code to suspicious requests is not often unique. This value always vary between the 4xx and 5xx status-codes,

Another advantage over the vector data-structure is that the hash-table will have a finite size The amount of server response Status-Codes is a discrete set of values. While the vector that has no defined size.

Chapter 8

Conclusion

8.1 System Efficiency

The system's temporal efficiency makes it a fast on-line analysis tool. It would be necessary to have the system hooked on a real environment to measure the server response time impact on the IDS's stateful analysis. Having an average testing time of $4,289\mu s$, the system is capable of meeting close to real-time constraints. Real-time constraints are met considering that the additional constant timings added on by network data transport and server response timing do not impact the overall IDS's client request service phase.

Efficiency in terms of false-positives or false-negatives is greatly dependant on the training data provided by the user. Not only the training data but also the precision while selecting accurate HTTP request features and temporal patterns. A tool for eliciting HTTP request features was developed for this project but none for temporal patterns. A tool to assist this process would be of great helper to a system administrator using this IDS.

A disappointing fact is that most commercial application-level IDS do not implement stateful inspection. If they do, it is not based on the correlation of server events with associated suspicious requests. The reason behind is that to achieve real-time analysis and prevention, the IDS cannot depend on the response that it will "eventually" receive from the server.

8.2 Implementation techniques

As for implementation, C++ is a powerful language. Many OS components can be accessed directly through the GNU C libraries, which cannot with many others like Java. This enables the accurate measurement of the system in terms of nanoseconds.

The amount of available third party software, with an object-oriented abstraction is abundant. Such available libraries written under C++ as XML parser engines and regular expression processors. C++'s STL saves the programmers a great deal of work during debugging. Since container, e.g. vector, overflows are easier to identify than without the STL.

8.3 Closing Statement

I thought this day would never come. It has been fun . . .

Appendix A

Definitions and Abbreviations

A.1 Definitions

Paradigm, is defined under the given context as a model or a pattern of an information system.

Proxy, The agency for another who acts through the agen

Source Entity, is defined as the compound object formed of a host IP address, an alarm number and an HTTP Web-Server response Status-Code. Its components are called Source Entity Items. This term is useful since, it is not reliable to classify intrusions only by who is originating them i.e. the host computer's IP address. IP addresses can easily be spoofed, and there are several methods of taking control of a remote computer and making it act as a zombie. Considering these facts, it is also reliable to classify instrusions by which request is being sent.

Spoof, according to WordNet (r) 2.0 [20], to make a parody of.

Zombie, a computer that has been taken over by an intruder that performs actions on the intruders behalf.

A.2 Abbreviations

ART	Adaptive Resonance Theory
API	Application Programming Interface
ASP	Active Server Pages
ASP.NET	Active Server Pages under the .NET framework
BNF	Backus-Naur Form
CGI	The Common Gateway Interface
CIDF	The Common Intrusion Detection Framework
CMI	The Console Manager Interface
CPM	Coulomb Potential Model
DMZ	Demilitarized Zone
DoS	Denial of Service
FANNC	A Fast Adaptive Neural Network Classifier
FDL	Feature Description Language
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection Systems
IIS	Internet Information Server
NIDS	Network Intrusion Detection System
OS	Operating System
PDL	Program Description Language
PHP	PHP: Hypertext Preprocessor (HTML - embedded scripting language)
POSIX	Portable Operating System Interface (IEEE Standard 1003.1)
RAM	Random Access Memory
SQL	Structured Query Language
SSL	Secure Socket Layer
STL	Standard Template Library
TCP/IP	Transmission Control Protocol/Internet Protocol
TPDL	Temporal Pattern Description Language
URI	Uniform Resource Identifier
W3	alias for WWW
WWW	World Wide Web
XML	eXtensible Markup Language

Appendix B

Sample Configuration file

Here is a sample configuration file of the proposed IDS. This file is read by the system at startup, and then all relevant initializations are performed.

```
# This line is a comment.
# W3IDS configuration file
# Fernando Alvarez 2004-2005

[Kernel]
client_threads = 5
neural_nets = 4
listen_port = 8080
server_port = 80
manager_port = 1998

# File list must be as big as the number of neural
# networks specified above. File names should be
# comma separated.
training_data_list = sql.txt,xss.txt,scan.txt,phony.txt

# Specifies the increments that will be applied each
# time a suspicious source entity persists with malevolent
# data. 2^n where n = n + delay_increments.
delay_increments = 2

# Maximum delay time (s) before a source is transmitted to
# the forbidden source entity list.
max_delay = 256

[Stateless]
header.Agent.size
header.Connection.size
header.Agent.size = 4*body.size
request_line.Method.IDEN
request_line.URI.ocurrences(‘‘%00’’)
header.Content-Length.IDEN

[Stateful]
NETID.Server-Error.4xx > 100
NETID.Client-Error.403 > 100
IP.Redirection.3xx > 20
IP.Client-Error.400 > 10
IP.Client-Error.401 > 10
NETID.Server-Error.404 > 30

[Blacklist]
request.URI.regexp(‘‘*/etc/passwd*’’)
body.regexp(‘‘*=*/etc/passwd*’’)
header.Cookie.size > 1024
header.Cookie.regexp(‘‘*’DELETE*’’)
```

Appendix C

FANNC Library

C.1 Overview

For a detailed description of the Neural Network's design refer to [6]. In this section of the appendices the programming interface to the library is shown. As one of the best teaching methods is, "learn by example". Here is some sample code on how the library can be used. The following code segment is provided in C++.

```
#include <vector>
#include ‘‘fannc.hpp’’

#define DEF_WIDTH   0.03f
#define H_BIAS      0.25f
#define DELTA       0.01f
#define L_THRESHOLD 0.70f
#define MAX_ERROR   0.01f

int main(void)
{
  vector<double> v_in(4, 0.034f);
  vector<double> v_out(1, 1.0);

  FANNCNetwork * fannc_net = new FANNCNetwork( sigmoid,
                                                DEF_WIDTH,
                                                H_BIAS,
                                                DELTA,
                                                L_THRESHOLD,
                                                MAX_ERROR );
  fannc_net->train(v_in, v_out);
  fannc_net->print_network();

  delete fannc_net;
  return 0;
}
```

An input and output training instance is needed. The input is initialized as four components, all of the equal to 0,034. The output has only one component initialized to one. The network is created, and then can either be trained or tested. For it to be of any use it must be trained first.

C.2 FANNC Class List

C.2.1 FANNC Library Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

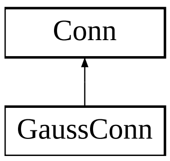
C.3 FANNC Class Documentation

C.3.1 Conn Class Reference

Neuron(p. 115) normal connection.

`#include <conn.hpp>`

Inheritance diagram for Conn::



Public Member Functions

- **Conn** (unsigned int _neuron=PLUS_INF, double _weight=MINUS_INF)

  Constructor.

- unsigned int **get_neuron** () const

  Get the destiny neuron's layer position.

- double **get_weight** () const

  Get the connection's weight.

- void **set_weight** (double _weight)

  Set the connection's weight.

Protected Attributes

- unsigned int **neuron**

  The destiny neuron's layer position.

- double **weight**

  The connection's weight.

C.3.1.1 Constructor & Destructor Documentation

Conn::Conn (unsigned int *_neuron* = `PLUS_INF`, double *_weight* = `MINUS_INF`)

The constructor initializes the destiny neuron's position to infinity and the connection's weight to -infinity.

The documentation for this class was generated from the following files:

- conn.hpp
- conn.cpp

C.3.2 DataErrorException Class Reference

FANNC exception handling.

`#include <errors.hpp>`

Public Member Functions

- **DataErrorException** (const std::string _pmsg="Error: Undefined", unsigned int _-code=UNDEF_ECODE)

  Constructor.

- const std::string **get_pmsg** () const

  Get the exception message pmsg.

Protected Attributes

- const std::string **pmsg**

  The exception message.

- unsigned int **code**

  The error message code.

C.3.2.1 Constructor & Destructor Documentation

DataErrorException::DataErrorException (const std::string *_pmsg* = `"Error: Undefined"`**, unsigned int *_code* =** `UNDEF_ECODE`**)**

Default constructor throws an undefined message with undefined error code.

Parameters:

_pmsg The exception message.

_code The error message code.

The documentation for this class was generated from the following files:

- errors.hpp
- errors.cpp

C.3.3 FANNCDOM Class Reference

The FANNC XML document abstraction.

`#include <fannc_dom.hpp>`

Public Member Functions

- **FANNCDOM** (**FANNCNetwork** &_net, const char ∗_xml_file="network.xml")

  Constructor.

- int **stream_net** (const char ∗_xml_file="std_out")

  Save or Print the referenced FANNCNEtwork.

- int **load_net** (const char ∗_xml_file="network.xml")

  Load an **FANNCNetwork**(p. 106) *from file. TODO!*

C.3.3.1 Detailed Description

The FANNC XML document abstraction. This class depends on the C++ Xerces Library. This library is distributed freely by Apache.org.

C.3.3.2 Constructor & Destructor Documentation

FANNCDOM::FANNCDOM (FANNCNetwork & *_net*, const char ∗ *_xml_file* = `"network.xml"`**)**

The constructor initializes the reference to the **FANNCNetwork**(p. 106) object and sets the file file.

Parameters:

_net The reference to **FANNCNetwork**(p. 106)

_xml_file The XML file to save the networks to.

C.3.3.3 Member Function Documentation

int FANNCDOM::load_net (const char ∗ *_xml_file* = `"network.xml"`**)**

TODO! Load an existing neural network to the from a file.

Parameters:

_xml_file The file-name where the Neural Network resides.

int FANNCDOM::stream_net (const char ∗ *_xml_file* = `"std_out"`**)**

Save or Print the referenced FANNCNEtwork.

Parameters:

_xml_file The file name. By default all serialization is done to the standard output.

The documentation for this class was generated from the following file:

- fannc_dom.hpp

C.3.4 FANNCNetwork Class Reference

Main FANNC Network class.

`#include <fannc.hpp>`

Public Member Functions

- **FANNCNetwork** (double(∗)(double), double _alpha=DEF_WIDTH, double _bias=H_-BIAS, double _delta=DELTA, double _threshold=L_THRESHOLD, double _max_-error=MAX_ERROR)

  Default constructor.

- ~**FANNCNetwork** ()
- std::vector< **InputNeuron** > **get_inputlayer** () const

  Get a copy of the Input Layer.

- std::vector< **IICNeuron** > **get_iiclayer** () const

  Get a copy of the Internal Input Classification Layer.

- std::vector< **IOCNeuron** > **get_ioclayer** () const

  Get a copy of the Internal Output Classification Layer.

- std::vector< **OutputNeuron** > **get_outputlayer** () const

  Get a copy of the Output Layer.

- **FANNCDOM** ∗ **get_XMLDoc** () const

Get a pointer to the XML document.

- double **get_dalpha** () const

  Get the default responsive width.

- double **get_bias** () const

  Get the hidden layer bias.

- double **get_delta** () const

  Get the Gaussian center adjustment step.

- double **get_threshold** () const

  Get the leakage competition threshold.

- double **get_merror** () const

  Get the maximum allowable error.

- void **train** (std::vector< double > &_trainInput, std::vector< double > &_expected-Output)

  Start the training algorithm.

- void **test** (std::vector< double > &_testInput, std::vector< double > &_actualOutput)

  Start the testing algorithm.

- int **save_network** (const char ∗_xmlFile)

  Saves the neural network state to XML Doc.

- int **load_network** (const char ∗_fileName)

  Reads the neural networks values from a file.

- int **print_network** ()

  Prints the neural network to standard output.

Friends

- class **FANNCDOM**

  XML Serializer must have access to the layers.

C.3.4.1 Detailed Description

A neural network one-pass learning algorithm with incremental learning ability. This algorithm uses supervised training.

C.3.4.2 Constructor & Destructor Documentation

FANNCNetwork::FANNCNetwork (double(∗)(double), double *_alpha* = `DEF_WIDTH`**, double *_bias* =** `H_BIAS`**, double *_delta* =** `DELTA`**, double *_threshold* =** `L_THRESHOLD`**, double *_max_error* =** `MAX_ERROR`**)**

Default constructor.

Parameters:

The	activation function.
The	default responsive width.
The	hidden layer bias.
The	Gaussian center adjustment step.
The	leakage competition threshold.
The	maximum allowable error.

FANNCNetwork::~FANNCNetwork ()

Default destructor clears out all of the network's layers.

C.3.4.3 Member Function Documentation

int FANNCNetwork::load_network (const char ∗ *_fileName*)

Reads the neural networks values from a file.

Parameters:

_fileName	The file-name to load network from.

int FANNCNetwork::print_network ()

Prints the neural network to standard output.

int FANNCNetwork::save_network (const char ∗ *_xmlFile*)

Saves the neural network state once it has been trained.

Parameters:

_xmlFile File-name to store XML document.

void FANNCNetwork::test (std::vector< double > & *_testInput*, std::vector< double > & *_actualOutput*)

Classification of a given input instance generating an actual output vector. NOTE: the network must first be trained.

Parameters:

_testInput The input instance.

_actualOutput The actual output instance.

void FANNCNetwork::train (std::vector< double > & *_trainInput*, std::vector< double > & *_expectedOutput*)

Start the training algorithm.

Parameters:

_trainInput The input instance.

_expectedOutput The expected output instance.

The documentation for this class was generated from the following files:

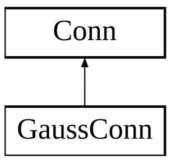
- fannc.hpp
- fannc.cpp

C.3.5 GaussConn Class Reference

Neuron(p. 115) Gaussian connection.

`#include <conn.hpp>`

Inheritance diagram for GaussConn::



Public Member Functions

- **GaussConn** (unsigned int _neuron, double _theta, double _alpha)

  Constructor.

- double **get_theta** () const

  Get the weight's center.

- double **get_alpha** () const

  Get the weight's width.

- void **set_theta** (double _theta)

  Set the weight's center.

- void **set_alpha** (double _alpha)

  Set the weight's width.

C.3.5.1 Constructor & Destructor Documentation

GaussConn::GaussConn (unsigned int *_neuron*, double *_theta*, double *_alpha*)

The constructor initializes the neuron's position, the weight's center and width.

Parameters:

_neuron The destiny neuron's layer position.

_theta The Gaussian weight's center.

_alpha The Gaussian weight's width.

The documentation for this class was generated from the following files:

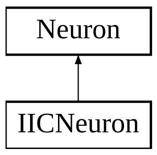
- conn.hpp
- conn.cpp

C.3.6 IICNeuron Class Reference

The FANNC Internal Input Classification neuron abstraction.

`#include <neuron.hpp>`

Inheritance diagram for IICNeuron::



Public Member Functions

- **IICNeuron** ()

  Constructor.

- bool **is_winner** () const

  Get the winner status of the neuron.

- void **set_winner** (bool _winner)

  Set the winner status of the neuron.

Protected Attributes

- **Conn output**

  The neuron's only single output connection.

- std::vector< **GaussConn** > **inputs**

  The neuron's Gaussian input connections.

Friends

- class **FANNCNetwork**

  FANNCNetwork(p. 106) *methods must access the neuron's content.*

- class **FANNCDOM**

  FANNCDOM(p. 105) *methods must access the neuron's content.*

The documentation for this class was generated from the following files:

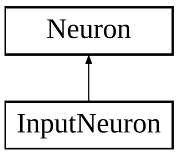
- neuron.hpp
- neuron.cpp

C.3.7 InputNeuron Class Reference

The FANNC Input neuron abstraction.

`#include <neuron.hpp>`

Inheritance diagram for InputNeuron::



Public Member Functions

- **InputNeuron** ()

  Constructor.

Protected Attributes

- std::vector< **GaussConn** > **outputs**

  The neuron's Gaussian output connections.

Friends

- class **FANNCNetwork**

  FANNCNetwork(p. 106) *methods must access the neuron's content.*

- class **FANNCDOM**

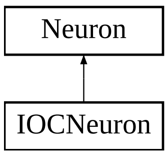
  FANNCDOM(p. 105) *methods must access the neuron's content.*

The documentation for this class was generated from the following files:

- neuron.hpp
- neuron.cpp

C.3.8 IOCNeuron Class Reference

The FANNC Internal Input Classification neuron abstraction.
`#include <neuron.hpp>`
Inheritance diagram for IOCNeuron::



Public Member Functions

- **IOCNeuron** ()

  Constructor.

- bool **is_winner** () const

  Get the neuron's winner status.

- double **get_cerror** () const

  Get the neuron's characteristic error.

- void **set_winner** (bool _winner)

  Set the neuron's winner status.

- void **set_cerror** (double _cError)

  Set the neuron's characteristic error.

Protected Attributes

- std::vector< **Conn** > **inputs**

  The neuron's normal input connections.

- std::vector< **Conn** > **outputs**

  The neuron's normal output connections.

- std::vector< **NeuronIndex** > **winnersMap**

  Optimization data-structure.

- std::priority_queue< **NeuronIndex**, std::vector< **NeuronIndex** >, std::greater< **NeuronIndex** > > **inputsMap**

  Optimization data-structure.

Friends

- class **FANNCNetwork**

  FANNCNetwork(p. 106) *methods must access the neuron's content.*

- class **FANNCDOM**

  FANNCDOM(p. 105) *methods must access the neuron's content.*

C.3.8.1 Member Data Documentation

std::priority_queue<NeuronIndex, std::vector<NeuronIndex>, std::greater<NeuronIndex> > IOCNeuron::inputsMap `[protected]`

Optimization priority queue which holds the IIC unit with maximum activation value, at the top, connected to this

std::vector<NeuronIndex> IOCNeuron::winnersMap `[protected]`

Optimization structure which holds all IIC winners connected to this.

The documentation for this class was generated from the following files:

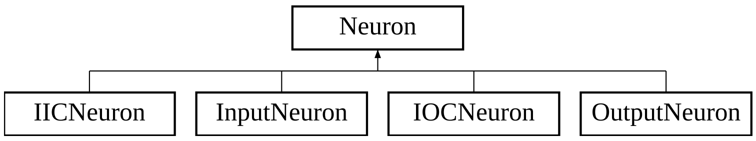
- neuron.hpp
- neuron.cpp

C.3.9 Neuron Class Reference

The FANNC base neuron abstraction.

`#include <neuron.hpp>`

Inheritance diagram for Neuron::



Public Member Functions

- **Neuron** (unsigned int _type=UNDEF_LAYER)

  Constructor.

- unsigned int **get_type** () const

  Get the neuron type.

- double **get_value** () const

  Get the activation value of the neuron.

- void **set_value** (double _value)

  Set the activation value of the neuron.

C.3.9.1 Constructor & Destructor Documentation

Neuron::Neuron (unsigned int *_type* = `UNDEF_LAYER`)

Constructor for neuron base class.

Parameters:

_type

C.3.9.2 Member Function Documentation

unsigned int Neuron::get_type () const `[inline]`

Get the neuron type.

Returns:

type

double Neuron::get_value () const `[inline]`

Get the activation value of the neuron.

Returns:

value

void Neuron::set_value (double *_value*)

Set the activation value of the neuron.

Parameters:

_value

The documentation for this class was generated from the following files:

- neuron.hpp
- neuron.cpp

C.3.10 NeuronIndex Class Reference

This class is used for optimization purposes.

`#include <neuron_idx.hpp>`

Public Member Functions

- **NeuronIndex** (unsigned int _neuron, double _value)

  Constructor.

- unsigned int **get_neuron** () const

Get the winner's position.

- double **get_value** () const

  Get the winner's activation value.

- bool **operator<** (const **NeuronIndex** &a) const

  Overloaded operator less than.

- bool **operator>** (const **NeuronIndex** &a) const

  Overloaded operator greater than.

C.3.10.1 Detailed Description

Used for optimization purposes while retrieving the winners from the second layer competition and the IIC unit with maximum activation value connected to a given IOC unit

C.3.10.2 Constructor & Destructor Documentation

NeuronIndex::NeuronIndex (unsigned int *_neuron*, double *_value*)
The constructor initializes neuron and value.

Parameters:

_neuron IICLayer winner's position.

_value IICLayer winner's activation value.

C.3.10.3 Member Function Documentation

bool NeuronIndex::operator< (const NeuronIndex & *a*) const
Overloaded operator less than, helps in the prioritizing of NeuronIndex objects within their priority queue.

bool NeuronIndex::operator> (const NeuronIndex & *a*) const
Overloaded operator greater than, helps in the prioritizing of NeuronIndex objects within their priority queue.

The documentation for this class was generated from the following files:

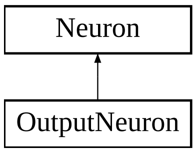
- neuron_idx.hpp
- neuron_idx.cpp

C.3.11 OutputNeuron Class Reference

The FANNC Output neuron abstraction.

`#include <neuron.hpp>`

Inheritance diagram for OutputNeuron::



Public Member Functions

- **OutputNeuron** ()

  Constructor.

Protected Attributes

- std::vector< **Conn** > **inputs**

  The neuron's normal input connections.

Friends

- class **FANNCNetwork**

  FANNCNetwork(p. 106) *methods must access the neuron's content.*

- class **FANNCDOM**

  FANNCDOM(p. 105) *methods must access the neuron's content.*

The documentation for this class was generated from the following files:

- neuron.hpp
- neuron.cpp

Appendix D

HTTP Parser Library

D.1 Overview

This is an incomplete and buggy HTTP Message parser. It is implemented after the specifications in [11]. It can read chunked requests in one string. It can also read multiple lines until the request is complete. Here is an example on how to use the library. In the next sub-section the publicly available API to the parser is given.

```
#include <iostream>
#include ‘‘http_req.hpp’’

const char sim_req = ‘‘GET /index.html HTTP/1.1\nHost: www.google.com\n\n’’;

int main(void)
{
  HTTPRequest * req = new HTTPRequest();

  req->parse(sim_req);

  if (req->complete())
     std::cout << req->to_string() << std::endl;
  else std::cout << ‘‘Sorry Bob!’’ << std::endl;

  delete req;
  return 0;
}
```

D.2 HTTP Class List

D.2.1 HTTP Parser Library Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

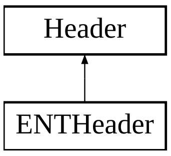
D.3 HTTP Class Documentation

D.3.1 ENTHeader Class Reference

The extended Entity **Header**(p. 122) class.

`#include <http_types.hpp>`

Inheritance diagram for ENTHeader::



Public Member Functions

- **ENTHeader** (const std::string _type_name="")

  Constructor.

- entity_header_base **get_base** () const

  Get the Entity **Header**(p. 122) *type.*

- void **set_base** (entity_header_base _base)

  Set the Entity **Header**(p. 122) *type.*

D.3.1.1 Constructor & Destructor Documentation

ENTHeader::ENTHeader (const std::string *_type_name* = "")

The constructor initializes the header name verifying that it is a valid header. Otherwise it is left as a null string.

The documentation for this class was generated from the following files:

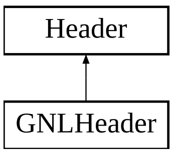
- http_types.hpp
- http_types.cpp

D.3.2 GNLHeader Class Reference

The extended General **Header**(p. 122) class.

`#include <http_types.hpp>`

Inheritance diagram for GNLHeader::



Public Member Functions

- **GNLHeader** (const std::string _type_name="")

  Constructor.

- general_header_base **get_base** () const

  Get the General **Header**(p. 122) *type.*

- void **set_base** (general_header_base _base)

  Set the General **Header**(p. 122) *type.*

D.3.2.1 Constructor & Destructor Documentation

GNLHeader::GNLHeader (const std::string *_type_name* = "")

The constructor initializes the header name verifying that it is a valid header. Otherwise it is left as a null string.

The documentation for this class was generated from the following files:

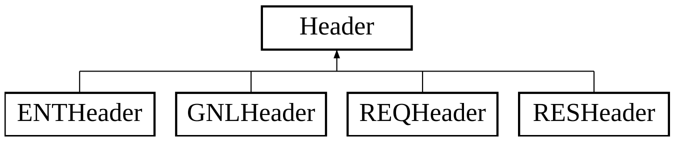
- http_types.hpp
- http_types.cpp

D.3.3 Header Class Reference

The Base Header class.

`#include <http_types.hpp>`

Inheritance diagram for Header::



Public Member Functions

- **Header** ()

  Constructor.

- std::string **get_type_name** () const

  Get the header name.

- std::string **get_content** () const

  Get the header content.

- void **set_type_name** (const std::string _type_name)

  Set the header name.

- void **set_content** (const std::string _content)

  Set the header content.

D.3.3.1 Constructor & Destructor Documentation

Header::Header ()

The constructor initializes both type_name and content to null strings.

The documentation for this class was generated from the following files:

- http_types.hpp
- http_types.cpp

D.3.4 HTTPRequest Class Reference

The HTTP Request class.

`#include <http_req.hpp>`

Public Member Functions

- **HTTPRequest** ()

  Constructor.

- ~**HTTPRequest** ()

  Destructor.

- bool **get_expect_body** () const

  Get the expect_body flag.

- bool **get_separator** () const

  Get the separator flag.

- **Method** $*$ **get_method** () const

  Get the request-line's method object.

- std::string $*$ **get_request_URI** () const

  Get the request-line's URI content.

- std::string $*$ **get_http_version** () const

Get the request-line's HTTP version content.

- std::vector< **GNLHeader** ∗ > **get_g_headers** () const

  Get a copy of all General Headers.

- std::vector< **REQHeader** ∗ > **get_r_headers** () const

  Get a copy of all Request Headers.

- std::vector< **ENTHeader** ∗ > **get_e_headers** () const

  Get a copy of all Entity Headers.

- std::vector< **Header** ∗ > **get_a_headers** () const

  Get a copy of all Add-on Headers.

- std::string ∗ **get_msg_body** () const

  Get the message-body content.

- bool **complete** ()

  Flag to indicate if request is completed.

- void **parse** (const char ∗_r_line)

  Start the parser engine.

- void **print_stats** ()

  Fetch request statistics.

- const char ∗ **to_string** ()

  Generate a single string HTTP Request.

D.3.4.1 Constructor & Destructor Documentation

HTTPRequest::HTTPRequest ()

The default constructor initializes all pointers to NULL. It also sets all boolean flags to false.

HTTPRequest::~HTTPRequest ()

Deletes the content of all pointers.

D.3.4.2 Member Function Documentation

bool HTTPRequest::complete ()

When a client sends a chunked request this flag is required to notify the parser engine when to stop.

void HTTPRequest::parse (const char ∗ *_r_line*)

Start the parser engine.

Parameters:

_r_line may be a whole request packed in one string or a series of them.

void HTTPRequest::print_stats ()

Print request statistics to standard output.

The documentation for this class was generated from the following files:

- http_req.hpp
- http_req.cpp

D.3.5 HTTPRequestException Class Reference

HTTP Request exception handling.

`#include <http_req.hpp>`

Public Member Functions

- **HTTPRequestException** (const std::string _pmsg="HTTPRequestException: Undefined\n")

  Constructor.

- const std::string **get_pmsg** () const

  Get the exception message pmsg.

Protected Attributes

- const std::string **pmsg**

  The exception message.

D.3.5.1 Constructor & Destructor Documentation

HTTPRequestException::HTTPRequestException (const std::string *_pmsg* = `"HTTPRequestException: Undefined\n"`**)**

Default constructor throws an undefined message.

Parameters:

_pmsg The exception message.

The documentation for this class was generated from the following files:

- http_req.hpp
- http_req.cpp

D.3.6 HTTPResponse Class Reference

The HTTP Response class.

`#include <http_res.hpp>`

Public Member Functions

- **HTTPResponse** ()

  Default constructor.

- **HTTPResponse** (const std::string _status)

  Constructor used to generate a response.

- ~**HTTPResponse** ()

  The default destructor.

- bool **complete** ()

  Check for chunked server responses.

- void **parse** (const char ∗_res)

  Begin response parsing.

- void **print_stats** ()

  Print response statistics to standard output.

- const char ∗ **to_string** ()

  Generate a single string HTTP Response.

D.3.6.1 Constructor & Destructor Documentation

HTTPResponse::HTTPResponse ()

The default constructor is used when a response is going to be received from a web-server.

HTTPResponse::HTTPResponse (const std::string *_status*)

The constructor can be used to generate a decoy response. This constructor can be used to mimic a web-server responding to request.

Parameters:

_status The HTTP response Status-Code.

HTTPResponse::~HTTPResponse ()

The default destructor is in charge of freeing all memory used by response object pointers.

D.3.6.2 Member Function Documentation

void HTTPResponse::parse (const char ∗ *_res*)

There are two modes of parsing that may be used. One for single line requests and another for multi-line requests.

Parameters:

_res The request line.

The documentation for this class was generated from the following files:

- http_res.hpp
- http_res.cpp

D.3.7 HTTPResponseException Class Reference

HTTP Response exception handling.

`#include <http_res.hpp>`

Public Member Functions

- **HTTPResponseException** (const std::string _pmsg="HTTPRequestException: Undefined\n")

  Constructor.

- const std::string **get_pmsg** () const

  Get the exception message pmsg.

Protected Attributes

- const std::string **pmsg**

  The exception message.

D.3.7.1 Constructor & Destructor Documentation

HTTPResponseException::HTTPResponseException (const std::string *_pmsg* = `"HTTPRequestException: Undefined\n"`**)**

Default constructor throws an undefined message.

Parameters:

_pmsg The exception message.

The documentation for this class was generated from the following files:

- http_res.hpp
- http_res.cpp

D.3.8 Method Class Reference

An HTTP request-line's method.

`#include <http_types.hpp>`

Public Member Functions

- **Method** (const std::string _type_name="")

  Constructor.

- method_base **get_type** () const

  Get the method type.

- std::string **get_type_name** () const

  Get the received method name.

- void **set_type** (method_base _type)

  Set the method type.

- void **set_type_name** (const std::string _type_name)

  Set the received method name.

D.3.8.1 Constructor & Destructor Documentation

Method::Method (const std::string *_type_name* = "")

The constructor initializes the method name verifying that it is a valid method. Otherwise it is left as a null string.

The documentation for this class was generated from the following files:

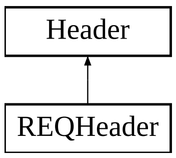
- http_types.hpp
- http_types.cpp

D.3.9 REQHeader Class Reference

The Request Headers class.

`#include <http_req.hpp>`

Inheritance diagram for REQHeader::



Public Member Functions

- **REQHeader** (const std::string _type_name="")

  Constructor.

- request_header_base **get_base** () const

  Get the type of header.

- void **set_base** (request_header_base _base)

  Set the type of header.

D.3.9.1 Constructor & Destructor Documentation

REQHeader::REQHeader (const std::string *_type_name* = "")
The default constructor will verify _type_name and set automatically the type The request header type.

The documentation for this class was generated from the following files:

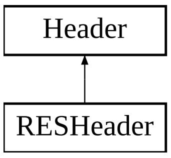
- http_req.hpp
- http_req.cpp

D.3.10 RESHeader Class Reference

The Response Headers class.

`#include <http_res.hpp>`

Inheritance diagram for RESHeader::



Public Member Functions

- **RESHeader** (const std::string _type_name="")

  Constructor.

- response_header_base **get_base** () const

  Get the type of header.

- void **set_base** (response_header_base _base)

  Set the type of header.

D.3.10.1 Constructor & Destructor Documentation

RESHeader::RESHeader (const std::string *_type_name* = "")

The default constructor will verify _type_name and set automatically the type The response header type.

The documentation for this class was generated from the following files:

- http_res.hpp
- http_res.cpp

D.3.11 STATUSCode Class Reference

The HTTP response-line Status-Code class.

`#include <http_res.hpp>`

Public Member Functions

- **STATUSCode** (const std::string _code_content="")

  Constructor.

- status_code_base **get_code** () const

  Get the Status-Code type.

- std::string **get_code_content** () const

  Get the Status-Code received content.

- std::string **get_comment** () const

  Get the response-line comment.

- void **set_code** (status_code_base _code)

  Set the Status-Code type.

- void **set_code_content** (const std::string _code_content)

  Set the Status-Code received content.

- void **set_comment** (const std::string _comment)

  Set the response-line comment.

D.3.11.1 Constructor & Destructor Documentation

STATUSCode::STATUSCode (const std::string *_code_content* = "")

The default constructor will verify _code_content and set automatically the type The Status-Code type.

The documentation for this class was generated from the following files:

- http_res.hpp
- http_res.cpp

Appendix E

FDL

E.1 Overview

The Feature Description Language is used to declare features that the Application-level HTTP IDS needs to process. These are processed by the system's neural networks. FDL forms part of the stateless validation carried out. The API to the language is available in this appendix. Bellow is a brief C++ program demonstrating how the language may be used

```
#include <iostream>
#include ‘‘http_req.hpp’’
#include ‘‘exp_lang.hpp’’

const char feature = ‘‘header.Host.size’’;
const char sim_req = ‘‘GET /index.html HTTP/1.1\nHost: www.google.com\n\n’’;

int main(void)
{
  HTTPRequest * req = new HTTPRequest();
  ExpLangObject * e = new ExpLangObject(feature);

  req->parse(sim_req);

  if (req->complete())
    std::cout << e->extract(req) << std::endl;
  else std::cout << ‘‘Sorry Bob!’’ << std::endl;

  delete e;
  delete req;
  return 0;
}
```

E.2 FDL Class List

E.2.1 Feature Description Language Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

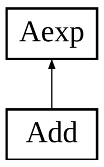
E.3 FDL Class Documentation

E.3.1 Add Class Reference

FDL addition expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Add::



Public Member Functions

- **Add** (**Aexp** ∗a1_, **Aexp** ∗a2_)

  Constructor.

- virtual double **A** () const

  Return the expressions value.

E.3.1.1 Detailed Description

FDL addition expression. Inherits virtual functions from **Aexp**(p. 135).

See also:
Aexp(p. 135)

E.3.1.2 Constructor & Destructor Documentation

Add::Add (Aexp ∗ *a1_*, Aexp ∗ *a2_*)
Perform the addition of the two operands (a1_ + a2_).

Parameters:

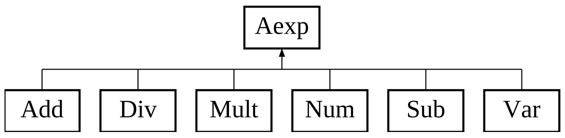
a1_	The first operand, left.
a2_	The second operand, right.

The documentation for this class was generated from the following files:

- exp_lang.hpp
- exp_lang.cpp

E.3.2 Aexp Class Reference

FDL Arithmetic Expression.
`#include <exp_lang.hpp>`
Inheritance diagram for Aexp::



Public Member Functions

- virtual double **A** () const =0

  Return the expressions value.

Protected Member Functions

- **Aexp** (HTTPRequest ∗request_)

  Constructor.

Protected Attributes

- HTTPRequest ∗ **request**

  Pointer to associated HTTPRequets object.

E.3.2.1 Constructor & Destructor Documentation

Aexp::Aexp (HTTPRequest ∗ *request_*) `[protected]`

Constructor of arithmetic expression initializes the request object.

See also:

HTTPRequest

E.3.2.2 Member Data Documentation

HTTPRequest∗ Aexp::request `[protected]`

Pointer to associated HTTPRequets object.

See also:

HTTPRequest

The documentation for this class was generated from the following files:

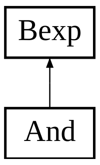
- exp_lang.hpp
- exp_lang.cpp

E.3.3 And Class Reference

FDL and expression.

`#include <exp_lang.hpp>`

Inheritance diagram for And::



Public Member Functions

- Constructor **And** (**Bexp** ∗b1_, **Bexp** ∗b2_)
- virtual bool **B** () const

  Return the expressions value.

E.3.3.1 Detailed Description

FDL and expression. Inherits virtual functions from **Bexp**(p. 138).

See also:
Bexp(p. 138)

E.3.3.2 Constructor & Destructor Documentation

And::And (Bexp ∗ *b1_*, Bexp ∗ *b2_*)
Perform the and of the two operands (b1_ & b2_).

Parameters:

b1_	The first operand, left.
b2_	The second operand, right.

The documentation for this class was generated from the following files:

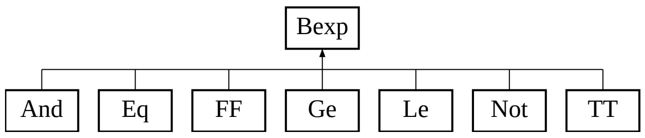
- exp_lang.hpp
- exp_lang.cpp

E.3.4 Bexp Class Reference

FDL Boolean Expressions.

`#include <exp_lang.hpp>`

Inheritance diagram for Bexp::



Public Member Functions

- virtual bool **B** () const =0

  Return the expressions value.

Protected Member Functions

- **Bexp** ()

  Constructor.

The documentation for this class was generated from the following files:

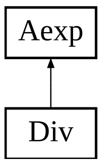
- exp_lang.hpp
- exp_lang.cpp

E.3.5 Div Class Reference

FDL division expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Div::



Public Member Functions

- **Div** (**Aexp** $*$, **Aexp** $*$)

  Constructor.

- virtual double **A** () const

  Return the expressions value.

E.3.5.1 Detailed Description

FDL division expression. Inherits virtual functions from **Aexp**(p. 135).

See also:
  Aexp(p. 135)

E.3.5.2 Constructor & Destructor Documentation

Div::Div (Aexp $*$, Aexp $*$)
  Perform the division of the two operands (a1_ / a2_) .

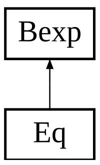
Parameters:
  a1_ The first operand, left.
  a2_ The second operand, right.

The documentation for this class was generated from the following files:

- exp_lang.hpp
- exp_lang.cpp

E.3.6 Eq Class Reference

FDL equality expression.
  `#include <exp_lang.hpp>`
  Inheritance diagram for Eq::



Public Member Functions

- **Eq** (**Aexp** ∗a1_, **Aexp** ∗a2_)

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

E.3.6.1 Detailed Description

FDL equality expression. Inherits virtual functions from **Bexp**(p. 138).

See also:
Bexp(p. 138)

E.3.6.2 Constructor & Destructor Documentation

Eq::Eq (Aexp ∗ *a1_*, Aexp ∗ *a2_*)
Perform the equality of the two operands (a1_ == a2_).

Parameters:

a1_	The first operand, left.
a2_	The second operand, right.

The documentation for this class was generated from the following files:

- exp_lang.hpp
- exp_lang.cpp

E.3.7 ExpLangObject Class Reference

FDL Parser Engine.

```
#include <exp_lang.hpp>
```

Public Member Functions

- **ExpLangObject** (const char ∗inst_)

  Constructor.

- double **extract** (HTTPRequest ∗request_)

  Initialize parse engine.

E.3.7.1 Detailed Description

Each object of the ExpLangObject class may be seen as an individual parser engine.

E.3.7.2 Constructor & Destructor Documentation

ExpLangObject::ExpLangObject (const char ∗ *inst_*)
Initializes exp to inst_

Parameters:
inst_ The user-declared expression.

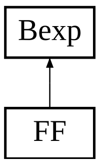
E.3.7.3 Member Function Documentation

double ExpLangObject::extract (HTTPRequest ∗ *request_*)
Initialize parse engine. request_ The variable lookup table.
The documentation for this class was generated from the following files:

- exp_lang.hpp
- exp_lang.cpp

E.3.8 FF Class Reference

FDL false expression.
`#include <exp_lang.hpp>`
Inheritance diagram for FF::



Public Member Functions

- **FF** ()

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

The documentation for this class was generated from the following files:

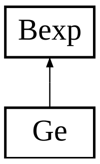
- exp_lang.hpp
- exp_lang.cpp

E.3.9 Ge Class Reference

FDL greater than expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Ge::



Public Member Functions

- **Ge** (**Aexp** ∗a1_, **Aexp** ∗a2_)

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

E.3.9.1 Detailed Description

FDL greater than expression. Inherits virtual functions from **Bexp**(p. 138).

See also:
Bexp(p. 138)

E.3.9.2 Constructor & Destructor Documentation

Ge::Ge (Aexp ∗ *a1_*, Aexp ∗ *a2_*)
Perform the greater than of the two operands (a1_ > a2_).

Parameters:

a1_	The first operand, left.
a2_	The second operand, right.

The documentation for this class was generated from the following files:

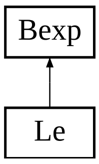
- exp_lang.hpp
- exp_lang.cpp

E.3.10 Le Class Reference

FDL less than expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Le::



Public Member Functions

- **Le** (**Aexp** $*$a1_, **Aexp** $*$a2_)

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

E.3.10.1 Detailed Description

FDL less than expression. Inherits virtual functions from **Bexp**(p. 138).

See also:
Bexp(p. 138)

E.3.10.2 Constructor & Destructor Documentation

Le::Le (Aexp $*$ *a1_*, Aexp $*$ *a2_*)

Perform the less than of the two operands (a1_ < a2_).

Parameters:

a1_ The first operand, left.

a2_ The second operand, right.

The documentation for this class was generated from the following files:

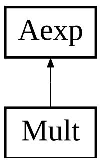
- exp_lang.hpp
- exp_lang.cpp

E.3.11 Mult Class Reference

FDL multiplication expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Mult::



Public Member Functions

- **Mult** (**Aexp** ∗a1_, **Aexp** ∗a2_)

  Constructor.

- virtual double **A** () const

  Return the expressions value.

E.3.11.1 Detailed Description

FDL multiplication expression. Inherits virtual functions from **Aexp**(p. 135).

See also:
Aexp(p. 135)

E.3.11.2 Constructor & Destructor Documentation

Mult::Mult (Aexp ∗ *a1_*, Aexp ∗ *a2_*)

Perform the multiplication of the two operands (a1_ ∗ a2_).

Parameters:

a1_ The first operand, left.

a2_ The second operand, right.

The documentation for this class was generated from the following files:

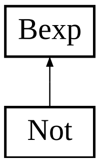
- exp_lang.hpp
- exp_lang.cpp

E.3.12 Not Class Reference

FDL not expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Not::



Public Member Functions

- **Not** (**Bexp** ∗b_)

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

E.3.12.1 Detailed Description

FDL not expression. Inherits virtual functions from **Bexp**(p. 138).

See also:
Bexp(p. 138)

E.3.12.2 Constructor & Destructor Documentation

Not::Not (Bexp ∗ *b_*)

Perform the not of the operand (!b).

Parameters:
b_ The operand.

The documentation for this class was generated from the following files:

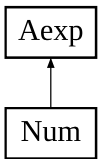
- exp_lang.hpp
- exp_lang.cpp

E.3.13 Num Class Reference

FDL numerical expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Num::



Public Member Functions

- **Num** (const char ∗value_)

  Constructor.

- virtual double **A** () const

  Return the expressions value.

E.3.13.1 Detailed Description

FDL numerical expression. Inherits virtual functions from **Aexp**(p. 135).

See also:
Aexp(p. 135)

E.3.13.2 Constructor & Destructor Documentation

Num::Num (const char ∗ *value_*)

Constructor of numerical expression

Parameters:
value_ The numerical value.

The documentation for this class was generated from the following files:

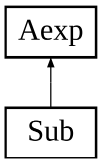
- exp_lang.hpp
- exp_lang.cpp

E.3.14 Sub Class Reference

FDL subtraction expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Sub::



Public Member Functions

- **Sub** (**Aexp** ∗a1_, **Aexp** ∗a2_)

  Constructor.

- virtual double **A** () const

  Return the expressions value.

E.3.14.1 Detailed Description

FDL subtraction expression. Inherits virtual functions from **Aexp**(p. 135).

See also:

Aexp(p. 135)

E.3.14.2 Constructor & Destructor Documentation

Sub::Sub (Aexp ∗ *a1_*, Aexp ∗ *a2_*)

Perform the subtraction of the two operands (a1_ - a2_).

Parameters:

a1_	The first operand, left.
a2_	The second operand, right.

The documentation for this class was generated from the following files:

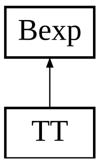
- exp_lang.hpp
- exp_lang.cpp

E.3.15 TT Class Reference

FDL true expression.

`#include <exp_lang.hpp>`

Inheritance diagram for TT::



Public Member Functions

- **TT** ()

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

The documentation for this class was generated from the following files:

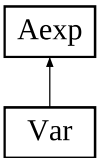
- exp_lang.hpp
- exp_lang.cpp

E.3.16 Var Class Reference

FDL variable expression.

`#include <exp_lang.hpp>`

Inheritance diagram for Var::



Public Member Functions

- **Var** (const char ∗name_, HTTPRequest ∗request_)

  Constructor.

- virtual double **A** () const

  Return the expressions value.

E.3.16.1 Detailed Description

FDL variable expression. Inherits virtual functions from **Aexp**(p. 135).

See also:
Aexp(p. 135)

E.3.16.2 Constructor & Destructor Documentation

Var::Var (const char ∗ *name_*, HTTPRequest ∗ *request_*)
Constructor of variable expression. The lookup "table" with possible values is the associated HTTP request object.

Parameters:

value_	The variable name.
request_	The associated request.

The documentation for this class was generated from the following files:

- exp_lang.hpp
- exp_lang.cpp

Appendix F

TPDL

F.1 Overview

F.2 Overview

The Temporal Patterns Description Language is used to declare correlations between between Source Entity Items. That is, how IP addresses, Alarm number and Web-Server response Status-Codes relate among each other. TPDL forms part of the stateful validation carried out. The API to the language is available in this appendix. Bellow is a brief C++ program demonstrating how the language may be used.

```
#include <iostream>
#include ‘‘http_req.hpp’’
#include ‘‘patt_lang.hpp’’

const char feature = ‘‘NETID.Client-Error.4xx > 4’’;
const char sim_req = ‘‘GET /index.html HTTP/1.1\nHost: www.google.com\n\n’’;

int main(void)
{
  SRCENTDat * current = new SRCENTDat(‘‘130.225.63.21’’, 2);
  current->set_srv_res(‘‘404’’);

  SRCENTBin * src_enities = new SRCENTBin();

  TMPPattern * p = new TMPPattern(feature, src_entities, current);

  std::cout << p->extract() std::endl;

  delete current;
  delete src_entities;
  delete p;

  return 0;
}
```

Notice that the above code would not do anything, since the Source Entity Container is empty. It must be filled up before with previous Source Entity Dat elements. An element is a group of Source Entity Items i.e. an IP address, an Alarm number and a Web-Server response Status-Code.

F.3 TPDL Class List

F.3.1 Temporal Pattern Description Language Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

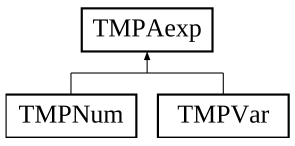
F.4 TPDL Class Documentation

F.4.1 TMPAexp Class Reference

TPDL Arithmetic Expression.

`#include <patt_lang.hpp>`

Inheritance diagram for TMPAexp::



Public Member Functions

- virtual int **A** () const =0

  Return the expressions value.

Protected Member Functions

- **TMPAexp** (SRCEntities ∗)

  Constructor.

Protected Attributes

- SRCEntities ∗ **srcents_bin**

  Pointer to associated SRCEntities object.

F.4.1.1 Constructor & Destructor Documentation

TMPAexp::TMPAexp (SRCEntities ∗) `[protected]`
Constructor of arithmetic expression initializes the srcents_bin object.

See also:
SRCEntitites

F.4.1.2 Member Data Documentation

SRCEntities∗ TMPAexp::srcents_bin `[protected]`
Pointer to associated SRCEntities object.

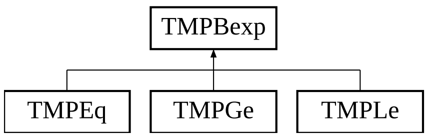
See also:
SRCEntities

The documentation for this class was generated from the following files:

- patt_lang.hpp
- patt_lang.cpp

F.4.2 TMPBexp Class Reference

TPDL Boolean Expressions.
`#include <patt_lang.hpp>`
Inheritance diagram for TMPBexp::



Public Member Functions

- virtual bool **B** () const =0

  Return the expressions value.

Protected Member Functions

- **TMPBexp** ()

  Constructor.

The documentation for this class was generated from the following files:

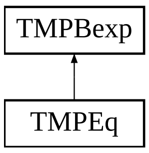
- patt_lang.hpp
- patt_lang.cpp

F.4.3 TMPEq Class Reference

TPDL equality expression.

`#include <patt_lang.hpp>`

Inheritance diagram for TMPEq::



Public Member Functions

- **TMPEq (TMPAexp** ∗, **TMPAexp** ∗)

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

F.4.3.1 Detailed Description

TPDL equality expression. Inherits virtual functions from Bexp.

See also:
Bexp

F.4.3.2 Constructor & Destructor Documentation

TMPEq::TMPEq (TMPAexp $*$, TMPAexp $*$)
Perform the equality of the two operands (a1_ == a2_).

Parameters:

a1_ The first operand, left.

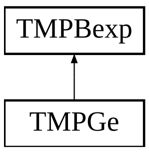
a2_ The second operand, right.

The documentation for this class was generated from the following files:

- patt_lang.hpp
- patt_lang.cpp

F.4.4 TMPGe Class Reference

TPDL greater than expression.
`#include <patt_lang.hpp>`
Inheritance diagram for TMPGe::



Public Member Functions

- **TMPGe** (**TMPAexp** $*$, **TMPAexp** $*$)

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

F.4.4.1 Detailed Description

TPDL greater than expression. Inherits virtual functions from Bexp.

See also:
Bexp

F.4.4.2 Constructor & Destructor Documentation

TMPGe::TMPGe (TMPAexp $*$, TMPAexp $*$)
Perform the greater than of the two operands (a1_ > a2_).

Parameters:

a1_ The first operand, left.

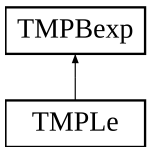
a2_ The second operand, right.

The documentation for this class was generated from the following files:

- patt_lang.hpp
- patt_lang.cpp

F.4.5 TMPLe Class Reference

TPDL less than expression.
`#include <patt_lang.hpp>`
Inheritance diagram for TMPLe::



Public Member Functions

- **TMPLe** (**TMPAexp** $*$, **TMPAexp** $*$)

  Constructor.

- virtual bool **B** () const

  Return the expressions value.

F.4.5.1 Detailed Description

TPDL less than expression. Inherits virtual functions from Bexp.

See also:
Bexp

F.4.5.2 Constructor & Destructor Documentation

TMPLe::TMPLe (TMPAexp ∗, TMPAexp ∗)
Perform the less than of the two operands (a1_ < a2_).

Parameters:

a1_ The first operand, left.

a2_ The second operand, right.

The documentation for this class was generated from the following files:

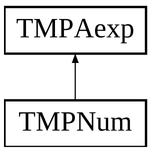
- patt_lang.hpp
- patt_lang.cpp

F.4.6 TMPNum Class Reference

TPDL numerical expression.

`#include <patt_lang.hpp>`

Inheritance diagram for TMPNum::



Public Member Functions

- **TMPNum** (const char ∗value_)

  Constructor.

- virtual int **A** () const

  Return the expressions value.

F.4.6.1 Detailed Description

TPDL numerical expression. Inherits virtual functions from Aexp.

See also:
Aexp

F.4.6.2 Constructor & Destructor Documentation

TMPNum::TMPNum (const char ∗ *value_*)
Constructor of numerical expression

Parameters:
*value_* The numerical value.

The documentation for this class was generated from the following files:

- patt_lang.hpp
- patt_lang.cpp

F.4.7 TMPPattern Class Reference

FDL Parser Engine.
`#include <patt_lang.hpp>`

Public Member Functions

- **TMPPattern** (const char ∗exp_)

  Constructor.

- bool **extract** (SRCEntities ∗srcents_bin_, SRCENTDat ∗srcent_)

  Initialize parse engine.

- std::string **get_type** () const

  Get the type of declared temporal pattern.

F.4.7.1 Detailed Description

Each object of the ExpLangObject class may be seen as an individual parser engine.

F.4.7.2 Constructor & Destructor Documentation

TMPPattern::TMPPattern (const char ∗ *exp_*)
Initializes exp to exp_

Parameters:
*exp_* The user-declared expression.

F.4.7.3 Member Function Documentation

bool TMPPattern::extract (SRCEntities ∗ *srcents_bin_*, SRCENTDat ∗ *srcent_*)
Initialize parse engine.

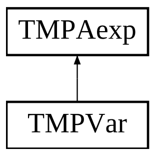
Parameters:
*srcents_bin_* The pointer to the Source Entity Container.
*srcent_* The pointer to the current Suspicious Source Entity content.

std::string TMPPattern::get_type () const `[inline]`
Get the type of declared temporal pattern. ("IP" | "ALARM")
The documentation for this class was generated from the following files:

- patt_lang.hpp
- patt_lang.cpp

F.4.8 TMPVar Class Reference

TPDL variable expression.
`#include <patt_lang.hpp>`
Inheritance diagram for TMPVar::



Public Member Functions

- **TMPVar** (const char ∗exp_, SRCEntities ∗srcents_bin_)

  Constructor.

- void **set_srcent** (SRCENTDat ∗)

  Set the Pointer to the Source Entity Container.

- virtual int **A** () const

    Return the expressions value.

F.4.8.1 Detailed Description

TPDL variable expression. Inherits virtual functions from Aexp.

See also:
Aexp

F.4.8.2 Constructor & Destructor Documentation

TMPVar::TMPVar (const char ∗ *exp_*, SRCEntities ∗ *srcents_bin_*)
Constructor of variable expression. The lookup "table" with possible values is the IDS's Suspicious Source Entity Container.

Parameters:

exp_ The variable name.

srcents_bin_ The pointer to the Source Entity Container.

The documentation for this class was generated from the following files:

- patt_lang.hpp
- patt_lang.cpp

Appendix G

Analysis Results

G.1 Overview

For all of the analysis carried out with the HTTP dissection tool, there was a set of 10 sample requests gathered. There was 10 anomal samples from each of the Web-Clients. For each of the signature-based misuse requests 10 samples of each type of attack were gathered. The signature-based misuse request were obtained from Nessus. Ethereal was then used to listen in the conversation between the Web-Server and Nessus.

For the anomal behavior request, there are two types of different requests; one during normal navigation and another during the interaction with the PHP scripting sub-system. The tool PHPMyMoney was used as a test sink for Nessus. PHPMyMoney does not only make heavy use of the PHP scripting sub-system, it also interact with MySQL Database-Server.

G.2 Tables

Browser	GET Usage	POST Usage	URL Range	Version
Firefox	100%		[1..56]	HTTP/1.1
Firefox PHP	90%	10%	[21..38]	HTTP/1.1
Opera	100%		[1..16]	HTTP/1.1
Opera PHP	90%	10%	[12..45]	HTTP/1.1
IE	100%		[1..7]	HTTP/1.1
IE PHP	90%	10%	[11..38]	HTTP/1.1
Mozilla	100%		[1..16]	HTTP/1.1
Mozilla PHP	90%	10%	[12..38]	HTTP/1.1

Table G.1: Content size and usage features extracted from sample anomal HTTP request Request-Lines.

Browser	Accept	Accept-Charset	Accept-Encoding	Accept-Language	Host
Firefox	99	30	12		[14..17]
Firefox PHP	[3..19]	30	12		17
Opera	117	54	38	2	17
Opera PHP	117	54	38	2	17
IE	164		13	17	17
IE PHP	[3..164]		13	17	17
Mozilla	[19..99]	30	12		9
Mozilla PHP	[3..99]	30	12		9

Table G.2: Content size feature extracted from sample anomal HTTP request Request Header fields.

Browser	If-Modified-Since	If-None-Match	Referer	TE	User-Agent
Firefox			24		84
Firefox PHP			[36..45]		84
Opera	29	20		42	84
Opera PHP	29	[19..21]	[36..45]	42	84
IE					84
IE PHP			[36..45]		84
Mozilla			26		84
Mozilla PHP	29		[28..37]		84

Table G.3: Content size feature extracted from sample anomal HTTP request Request Header fields.

Browser	Cache-Control	Connection
Firefox		10
Firefox PHP		10
Opera		14
Opera PHP		14
IE		10
IE PHP	8	10
Mozilla		5
Mozilla PHP		5

Table G.4: Content size features extracted from sample anomal HTTP request General Header fields.

Browser	Content-Length	Content-Type
Firefox		
Firefox PHP		
Opera		
Opera PHP	3	33
IE		
IE PHP		
Mozilla		
Mozilla PHP	2	33

Table G.5: Content size features extracted from sample anomal HTTP request Entity Header fields.

Browser	Cookie	Cookie2	Keep-Alive
Firefox	82		3
Firefox PHP	43		3
Opera			
Opera PHP	43	10	
IE			
IE PHP	43		
Mozilla			
Mozilla PHP	43		

Table G.6: Content size features extracted from samples anomal HTTP request Add-on Header fields.

Browser	Content Size	Max Segment	Repetitions
Firefox			
Firefox PHP			
Opera			
Opera PHP	116	e=	3
IE			
IE PHP			
Mozilla			
Mozilla PHP	[17..39]		

Table G.7: Content size and segment frequency analysis from sample anomal HTTP request bodies.

METHOD	XSS	PHP-SQL	GUESS	DoS
GET	76%	100%	100%	55%
OPTIONS				10%
TRACE				5%
POST	24%			5%
HEAD				5%
PUT				5%
DELETE				5%
COPY				5%
SEARCH				5%

Table G.8: Method distribution from signature-based misuse HTTP request Request-Lines.

	XSS	PHP-SQL	GUESS	DoS
URL Size	[10..144]	[11..113]	[3..22]	[1..9]
Max. Segment				
Repetitions				
Version	HTTP/1.1	HTTP/1.1	HTTP/1.1	Various Content

Table G.9: Content size and segment frequency analysis from sample signature-based misuse HTTP request Request-Lines.

	XSS	PHP-SQL	GUESS	DoS
Accept	67	67	67	67
Accept-Charset	18	18	18	18
User-Agent	34	34	34	34
Accept-Language	2	2	2	2
Authorization	30		30	
Host	21	17	21	21

Table G.10: Content size feature extracted from sample signature-based misuse HTTP request Request Header fields.

	XSS	PHP-SQL	GUESS	DoS
Connection	10	10	10	5
Pragma	8	8	8	8

Table G.11: Content size feature extracted from sample signature-based misuse HTTP request General Header fields.

	XSS	PHP-SQL	GUESS	DoS
Content-Length	3			
Content-Type	33			

Table G.12: Content size feature extracted from sample signature-based misuse HTTP request Entity Header fields.

	XSS	**PHP-SQL**	**GUESS**	**DoS**
Cookie	51	51		51
Keep-Alive	3			

Table G.13: Content size feature extracted from sample signature-based misuse HTTP request Add-on Header fields.

Bibliography

[1] Security in Computing, Charles P. Pfleeger, Shari L. Pfleeger, 2003

[2] The Poor Man's Guide to Computer Networks and Their Applications, Robin Sharp, April 2004

[3] Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection, Thomas H. Ptacek, Timothy N. Newsham, January 1998

[4] Improving Intrusion Detection Performance Using Keyword Selection and Neural Networks, Richard P. Lippmann, Robert K. Cunningham

[5] Application-Integrated Data Collection for Security Monitoring, Magnus Almgren, Ulf Lindqvist, October 2001

[6] A Fast Adaptive Neural Network Classifier, Zhihua Zhou, Shifu Chen, Zhaoqian Chen, 2000

[7] Anomaly Detection of Web-based Attacks, Christopher Kruegel, Giovanni Vigna, October 2003

[8] A Stateful Intrusion Detection System for World-Wide Web Servers, Giovanni Vigna, William Robertson

[9] A Lightweight Tool for Detecting Web Server Attacks, Magnus Almgren, Herve Debar

[10] SecurityTracker Statistics 2002, http://www.securitytracker.com/learn/statistics.html, April 2002

[11] Hypertext Transfer Protocol Specification, June 1999, http://www.w3.org/Protocols/rfc2616/rfc2616.html

[12] Intelliwall: intelligent firewall for applications, September 2004, http://www.beeware.net/#

[13] STAT Framework, http://www.cs.ucsb.edu/r̃sg/STAT/software/stat_framework.html

[14] SecureIIS Web Server Protection, http://www.eeye.com/html/products/secureiis/index.html

[15] Titan, http://www.flicks.com/titan/

[16] Mod_Security, http://www.modsecurity.org/

[17] McAfee Entercept Web Server Edition, Data sheet, 2004

[18] WebSTAT, http://ftp.ics.uci.edu/pub/websoft/wwwstat/

[19] AppliCure, http://www.applicure.com/index.php?p=products

[20] WordNet (r) 2.0, http://www.dict.org/

[21] Xerces C++ Parser, http://xml.apache.org/xerces-c/

[22] C++ Boost.Regex, http://www.boost.org/libs/regex/doc/introduction.html

[23] On-line Manual Section 3, clock_gettime, standard POSIX System

[24] clock_gettime usage, http://www.cis.temple.edu/ ingar-
gio/old/cis307f02/readings/unix3.html

Index