



UNIVERSITY OF ROME “LA SAPIENZA”

DEPARTMENT OF ENGINEERING

Specialization degree

COMPUTER SCIENCE,

ARCHITECTURE AND DISTRIBUTED SYSTEMS

# Modelling attack processes on the Internet, based on honeypot collected data

*Development of a high-interaction honeypot  
data processing system*



Assistant supervisors

Eric Alata

Mohamed Kaâniche

Karama Kanoun

Supervisor

Bruno Ciciani

Reviewer

Francesco Quaglia

Candidate

**Luigi Nardi**

### **Abstract**

Honeypots are more and more used to collect data on malicious activities on the internet and to better understand the strategies and techniques used by attackers to compromise target systems. Analysis and modeling methodologies are needed to support the characterization of attack processes based on the data collected from the honeypots.

This work is addressed to synthesize the attacks in order to better analyze them. It is a help to the research activity to have more tools for characterizing attack processes and scenarios, more details about what has been achieved during the observed period.

Security results will be obtained forging the existing data with the new retrieving application.

# Acknowledgements

The work presented in this book has been realized during a six-months internship at LAAS-CNRS, “Laboratoire d’Analyse et d’Architecture des Systèmes du Centre National de la Recherche Scientifique” of Toulouse, France, in the group TSF, “Tolérance aux fautes et Sécurité de Fonctionnement Informatique”.

A special thanks goes to my Italian supervisor professor Bruno Ciciani and to all the TSF staff, in particular to Mohamed Kaâniche and Karama Kanoun, research directors, for giving me the valuable opportunity to make this experience and take advantage of it.

All my gratitude is for Eric Alata, PhD Student at LAAS-CNRS, whose work and advices have been precious for my internship.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Research context . . . . .	5
1.2	High-interaction honeypot data processing system . . . . .	7
1.3	Structure of the thesis . . . . .	9
<b>2</b>	<b>Background and environment</b>	<b>11</b>
2.1	LAAS CNRS . . . . .	11
2.2	TSF group: Dependable Computing and Fault Tolerance . . . . .	12
2.3	Background . . . . .	14
2.3.1	Introduction . . . . .	14
2.3.2	Related work . . . . .	16
2.3.3	Architecture of the honeypot . . . . .	17
2.3.4	Experimental results . . . . .	21
2.3.5	Behavior of attackers . . . . .	26
2.3.6	Conclusion . . . . .	30
<b>3</b>	<b>Honeypot data processing system</b>	<b>32</b>
3.1	Architecture . . . . .	33
3.2	Use Case specification . . . . .	35
3.3	Class diagram . . . . .	37
3.3.1	General . . . . .	37
3.3.2	Server side . . . . .	37
3.3.3	Client side . . . . .	40
<b>4</b>	<b>Functionalities</b>	<b>50</b>
4.1	Dictionary attack . . . . .	50
4.1.1	What is a dictionary attack? . . . . .	50
4.1.2	Dictionary attack functionality . . . . .	51
4.1.3	Table representation of the data . . . . .	53
4.1.4	Graphic representation of the data . . . . .	54
4.2	Terminal information . . . . .	57
4.2.1	What is a terminal? . . . . .	57
4.2.2	Terminal information functionality . . . . .	59

<i>CONTENTS</i>	4
<b>5 Protocol communication</b>	<b>62</b>
5.1 General . . . . .	62
5.2 Dictionary attack . . . . .	64
5.3 Terminal information . . . . .	65
5.4 Data testing . . . . .	67
<b>6 SQL query processing</b>	<b>70</b>
6.1 General . . . . .	70
6.2 Dictionary attack . . . . .	72
6.3 Example: test . . . . .	74
6.3.1 Special cases . . . . .	78
6.4 Implementation environment: JDBC driver . . . . .	82
<b>7 Conclusion and future work</b>	<b>84</b>
<b>A Database structure</b>	<b>86</b>
<b>B Dictionary attack query</b>	<b>88</b>
<b>C Terminal information first interaction</b>	<b>91</b>
<b>D Terminal information second interaction</b>	<b>94</b>
<b>E Event channel</b>	<b>95</b>
<b>F Example of dictionary</b>	<b>97</b>
<b>Bibliography</b>	<b>100</b>

# Chapter 1

## Introduction

### 1.1 Research context

Computer security is the part of the computer science that deals with the protection of the computer systems against malicious threats affecting their availability, confidentiality and integrity. The computer protection against attacks is got acting on more levels: physical level, putting the servers in most secure places, and logical level that obliges the authentication and the authorization of the entity that represents the client of the system. There are many possible techniques of attack, for this reason it is necessary to use together different defensive techniques to protect a computer system, realizing more barriers between the attacker and the targeted host.

**Honeypot** In computer science, a honeypot (literally: “the honey’s pot”) it is a system, hardware or software component, used as “trap” or “bait” with goal of collecting data about malicious attacks against computer systems. The term “honeypot” it is often connected to Winnie the Pooh, a rag bear that it is often found in troubles because of his greediness toward the honey. It usually consists of a computer or a site that seems to belong to the network and to contain precious information, instead the reality is that it is well isolated and it has not sensitive or critical data.

The importance of a honeypot is the information that it gives on the nature and the frequency of the network attacks. The honeypots do not contain real information and therefore they should not be involved in any activity; surveys in opposite sense can reveal not authorized intrusions or malevolent in progress.

These systems can bring some risks to the network, and it must be handled with care. If they are not well protected an attacker could use them to access other systems.

**Types of honeypot** The Low-Interaction Honeypot are usually programs that emulate operating systems and services. These honeypots are simple to be installed and secure, but they can capture few information. In general these honeypots are daemons that create some virtual hosts on the network and they can associate to a host a multiple IP address. The hosts can be set in order to perform an arbitrary service, i.e., an operating system; a fundamental aspect for the study of the security is that these systems hide the real system between those virtual.

The High-Interaction Honeypot, instead, do not emulate anything: computers, applications or services are real. These honeypots are more complexes and they involve great risks, but they succeed in capturing more information. These honeypots allow the attacker to take possession of the host but under a strict control of the system administrator. All the operations performed through a remote shell, logins, execution of programs, ingoing and outgoing packets, local time, IP addresses, are carefully recorded in a central database that is subsequently analyzed.

The freedom left to the attacker depends from system to system.

The diffusion of the high-interaction honeypots is more limited respect to honeypot of low-interaction because of the risks. If the attacker takes possession of a host, he can use this resource to attack other network resources for example performing a botnet attack. This risk is avoid by the low-interaction honeypots because they do not leave a lot of liberty to the attackers.

**The CADHo project** The LAAS of Toulouse, France, is a national center of scientific research that carry out research on various themes like robotic, automatic, biotechnology, space, telecommunication, software technology and energy.

The group TSF of the LAAS treat the topic of computer security from twenty years through an international partnership with many centers of research and being part of projects like ReSIST that also has partner the university *La Sapienza* of Rome.

During the last decade, the clients of the Internet are facing a great variety of malevolent threats like virus, worm, denial of service attacks, attempts of phishing, etc. A lot of publications, offer useful information on the new vulnerabilities and on the security threats, with an indication of their gravities and an evaluation on the potential damages that can cause. These initiatives serve for studying the malwares and the propagation of the attacks in Internet, therefore, they give some information about the identification and the analysis of malevolent activity in Internet.

Nevertheless, such information are not enough to model the mechanisms of attack and to analyze their impact on the safety of the attacked machine. The project CADHo (Collection and Analysis of Dates from Honeypots)

inside ReSIST, in which we are involved, it is complementary to these initiatives and it is aimed to fill the gap of the insufficiency of the information.

The institute Eurécom at Nice is the in charged institute, on the project CADHo, of the installation of the greatest part of the honeypots. Until today around 40 honeypot platforms has been deployed in university and enterprises, 30 countries have involved in the 5 continents.

The project CADHo has different international partners and all the components of this group have installed some honeypots inside their domains. A second way for a honeypot installation is through organizations not directly connected with the project CADHo whether to have more security information on their system or for pure spirit of collaboration with the research, they let to install some honeypots in their own network. Evidently in the second case to install some high-interaction honeypots can involve some risks to the organization because the attacker is not left impotent but he is able to act. This is why the high-interaction honeypots do not have a big diffusion and why Eurécom only treats low-interaction honeypot. The high-interaction honeypots require more work and more constant attention.

Clearly the high-interaction honeypots are better to collect information on the attack activities once that the attacker has taken the control of the host and it tries to perform a intrusion process to increase his privilege level. Both the two systems high and low-interaction can bring some advantages in terms of research, both are studied in the project CADHo with the target of building models that characterize the behavior of the attackers and a support to the developing of new intrusion tolerant systems.

Only the group TSF of the project CADHo has deployed a high-interaction honeypot and shares with the whole group of research the obtained results. For the problems already mentioned it is not easy to find other organizations that want to deploy the high-interaction honeypots and overall that give the availability of some human resources<sup>1</sup>.

A lot of analyses have already been done and a lot of interesting conclusions are already been published these last years regarding the low-interaction honeypots. The high-interaction honeypots are a second stage inside the project CADHo and they are intensely working on them in these last months.

## 1.2 High-interaction honeypot data processing system

As already said, a honeypot is a machine connected to a network but that no one is supposed to use. If a connection occurs, it must be, at best an accidental error or, more likely, an attempt to attack the machine. The

---

<sup>1</sup>We do not speak here only of hardware resources, as in the case of low-interaction honeypot, because there is the need of a rigorous and constant checking of the attacker activities.

design of the high-interaction honeypot developed at LAAS in the context of the CADHo project is described in chapter 2.

Every new attempt targeting the honeypot is saved within a database that from some months is recording a huge quantity of raw data. To retrieve the information from this system, the team has written scripts. These scripts are written in shell programming and the visualization is on the console. In the current version of the prototype the analysis of the attack data, which is the core of the CADHo project, is not very easy. An operator must execute scripts and read the results without any possibility to have these data formatted in any other way he wants. The objective of our study is to develop a data processing system that is aimed to facilitate the interaction with the user and the analysis of the data recorded on the high-interaction honeypots.

More concretely, the aim of the project is to build a wrapper that allows to retrieve these information in a more efficient and effective way. The wrapper is the high-interaction honeypot data processing system, see figure 1.1, and it will have tools to synthesize data retrieved in a graphical way. The legacy system is the old program, composed of scripts, that allowed to

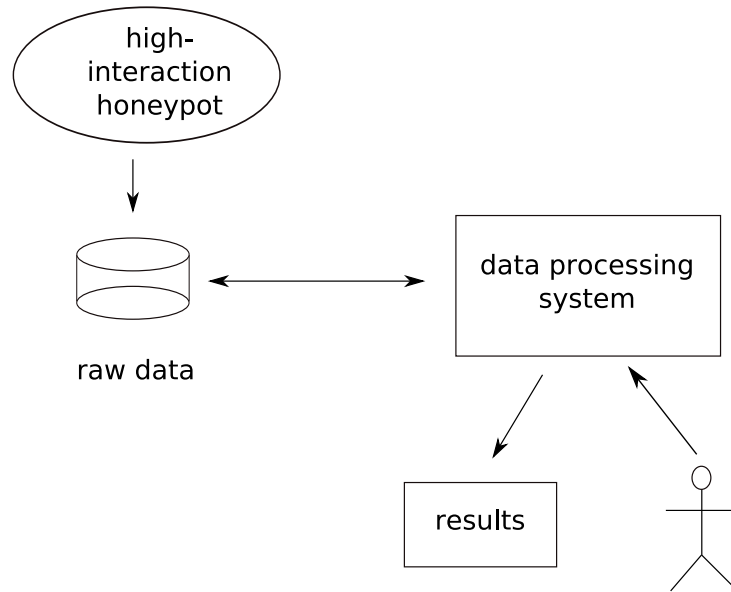


Figure 1.1: Honeypot data processing system

retrieve the information. The legacy system will continue to work normally without any additional constraints. Moreover we would like to develop an application that can be used from less skilled users. An application with graphic interface, that performs the complicated commands that an experienced operator performs on a shell, makes the system more usable.

The project is oriented to the develop of two modules that allow to retrieve database information in two different sets of data. In the future there will be an upgrade of the software, developing more modules beginning from the existing high-interaction honeypot data processing system.

Another important problem that the application should resolve is the performance on the server side. We need to develop powerful SQL scripts to get good performance<sup>2</sup> for each functionality, as always in an optic of improving the system usability.

The goal of this application is to help the research activity on high-interaction honeypots, providing a tool that allows to better analyze the recorded data.

### 1.3 Structure of the thesis

The chapters are structured in the following way:

- Chapter 2, Background and environment, contains a description of the organizational structures that have welcomed my job, the LAAS and more particularly the group TSF and a resume of the state of art on the honeypots at the begin of the high-interaction honeypot data processing system;
- Chapter 3, High-interaction honeypot data processing system, treats the analysis of the application developed, the architecture, layer and tier, the UML diagrams;
- Chapter 4, Functionalities, analyzes in detail the implemented functionalities and a global vision of the application;
- Chapter 5, Protocol communication, describes in detail the communication protocol, the data exchanged for each interaction between client and server. Here we also describe the data testing.
- Chapter 6, SQL query processing, shows all concerning the database part of the application and the dynamic mechanism to retrieve the SQL query scripts. Here we describe particularly the query of the functionality called Dictionary attack with an example on a fictitious database.
- Appendix A, Database structure, brings the structure of the honeypot database;
- Appendixes B, C, D, bring the scripts SQL for all the implemented functionalities;

---

<sup>2</sup>The results must be get with performance in the order of the second.

- Appendix E, Event channel, shows the implementation of a key part in the realization of the client side class diagram;
- Appendix F, Example of dictionary, represents an example of dictionary used by the attackers to perform a dictionary attack.

## Chapter 2

# Background and environment

### 2.1 LAAS CNRS

#### **History of the Laboratory for Analysis and Architecture of Systems**

LAAS-CNRS has been founded in July 1967 by Professor Jean Lagasse. It is one of the laboratories of CNRS, the French National Center for Scientific Research.

LAAS is closely associated to three universities in Toulouse, Université Paul Sabatier, Institut National Polytechnique, and Institut National des Sciences Appliquées. The members of LAAS are : research scientists at CNRS, faculty members, graduate and PhD students from these three universities, as well as engineers, technicians and administrative clerks at CNRS and the universities.

LAAS originated from a scientific background in electrical engineering. It has developed a wide international status in micro-electronics, in control theory, computer science and robotics. More generally, it has played an active role in the growth of Information and Communication Science and Technology in France. LAAS has contributed to the development of Toulouse as a well known international capital of aeronautics and space technologies.

Originally, the acronym LAAS meant Laboratory for Automation and its Applications to Space. It has become now the Laboratory for Analysis and Architecture of Systems. Its research topics, fundamental or applied, are focused on the study of complex systems, within multidisciplinary approaches. It has contributed to a variety of technological fields, from space and transport systems, to biotechnologies and health technologies, through telecommunications, software technology and energy.

## 2.2 TSF group: Dependable Computing and Fault Tolerance

TSF is the acronym of “*Tolérance aux fautes et Sécurité de Fonctionnement informatique*”. The research activities conducted by this group concern the dependability of computerized systems, which is the property allowing users of a system to justifiably trust the service it delivers. They take place in the context of a continuum between work aimed at advancement of knowledge and work in partnership with the socio-economic sector, meant at producing new services and products.

Dependability includes various properties, i.e., availability, reliability, integrity, confidentiality, maintainability, safety (against catastrophic failures) and security (against non-authorized access to information). Research deals with fault prevention, fault tolerance, fault removal and fault forecasting, and on the formulation of the basic concepts of dependability.

**Fault prevention** Fault prevention aims to avoid the occurrence or introduction of faults. It consists in avoiding design or manufacturing faults and preventing faults during operation. In this context, are developed methods for defining security policies, based on the identification of what properties should be implemented and what rules applications and organizations should abide by. These properties may be conflicting (for example, confidentiality and availability); the suggested policy should try to solve such conflicts in the best possible way.

Current work concerns medical and healthcare applications, which characteristics are their strong need for confidentiality, integrity and availability, and also accountability and privacy. The methods being developed are aimed at the definition of security policies adapted to the wide variety of organizations in which such applications will be implemented (hospitals, consulting rooms, health insurance offices, etc.).

**Fault Tolerance** Fault tolerance refers to a series of techniques used to allow a system to deliver correct service in spite of faults. Studies are centered on distributed software techniques for tolerating physical faults, design faults and deliberately malicious faults.

They deal with four main working areas:

- Protection of distributed applications on the Internet: servers tolerating both accidental and intentional faults are developed using, to the extent that it is possible, diversification of hardware platforms, operating systems and software;
- Use of the reflection principle for a transparent implementation of fault

tolerance: a multi-level approach has been defined and a platform developed using standard reflexive mechanisms;

- Wrapping of software executives to provide on-line checking of dependability properties. The latter are specified using a temporal logic modeling of the target software executives. This principle has been used for real-time microkernels and CORBA middleware.
- Protection of communications in a network of actuators: this study concerns the future real-time control systems for civil airplanes, and is carried out in collaboration with Airbus. Specific protection means based on error detecting codes have been proposed.

**Fault Removal** Fault removal aims to reduce the number and severity of faults. Research is focused on software testing. Recent studies have allowed an extension of the domain of application of statistical testing, which is a method for probabilistic generation of test inputs successfully applied in earlier work.

They concern three main areas:

- Test of reflexive software: a strategy of generic and incremental tests, based on decomposition of reflexive properties (reification, intercession and introspection), has been suggested in the context of a project with France Telecom;
- Study of the complementarity between test and proof: a method has been implemented for defining and carrying out tests using information obtained through formal proof;
- Tests with respect to safety properties: optimization heuristics are used to design test scenarios focused on dangerous faults.
- Testing of mobile-based applications and systems.

**Fault Forecasting** Fault forecasting aims at estimating the creation, existence and consequences of faults. Studies are concerned with forecasting the consequences of physical faults, design faults and malicious faults on system dependability. They cover both analytical and experimental evaluation.

Present work on analytical modeling is focused on developing hierarchical and compositional modeling approaches based on stochastic Petri nets, Markov chains and their extensions, in order to support the comparative evaluation of fault tolerant systems architectures during the design stage.

As far as experimental evaluation is concerned, two different aspects are studied: failure data analysis and controlled experiments. For the first one, work focuses on developing algorithms and procedures enabling us to use error logs from interconnected Unix and Windows systems to evaluate their

dependability. This study has been focused on the network of computers available at LAAS.

Controlled experiments concern: (a) characterization of failure modes of operating systems (Linux and Windows) and of CORBA middleware, and (b) the benchmarking of the dependability of computer systems. A conceptual framework for defining dependability benchmarks and benchmark prototypes have been achieved.

As regards malicious faults, research is focused on the development of analytical modeling approaches for the quantitative evaluation of security and experimental measurement approaches based on honeypots for the collection and analysis of real attacks on the Internet.

## 2.3 Background

This section presents the results obtained during this last months on the high-interaction honeypot from the CADHo project. All these results are discussed in the paper [1] and this is the beginning point for the development of the data processing system.

This section is organized as follows. Subsection 2.3.1 is just an introduction to the main concept. Subsection 2.3.2 discusses some existing techniques for developing high-interaction honeypots and the design rationales for the solution utilized in the paper. Subsection 2.3.3 describes the proposed solution. The lessons learned from the attacks observed over a period of almost 4.5 months are discussed in subsection 2.3.4. Finally, subsection 2.3.6 offers some conclusions as well as some ideas for future work.

### 2.3.1 Introduction

During the last decade, the Internet users have been facing a large variety of malicious threats and activities including viruses, worms, denial of service attacks, phishing attempts, etc. Several surveys and indicators, published at a regular basis, provide useful information about new vulnerabilities and security threats, with an indication of their estimated severities with respect to the potential damage that they might cause. On the other hand, several initiatives have been developed to monitor real world data related to malware and attacks propagation on the Internet. Among them, can be mentioned the Internet Motion Sensor project [2], CAIDA [3] and DShield [4]. These projects provide valuable information for the identification and analysis of malicious activities on the Internet. Nevertheless, such information is not sufficient to model attack processes and analyze their impact on the security of the targeted machines. The CADHo project [5] in which we are involved is complementary to these initiatives and is aimed at filling such a gap by carrying out the following activities:

- deploying and sharing with the scientific community a distributed platform of honeypots [6] that gathers data suitable to analyze the attack processes targeting a large number of machines connected to the Internet;
- validating the usefulness of this platform by carrying out various analysis, based on the collected data, to characterize the observed attacks and model their impact on security.

A honeypot is a machine connected to a network but that no one is supposed to use. In theory, no connection to or from that machine should be observed. If a connection occurs, it must be, at best an accidental error or, more likely, an attempt to attack the machine. Two types of honeypots can be distinguished depending on the level of interactivity that they offer to the attackers. Low-interaction honeypots do not implement real functional services. They emulate simple services which cannot be compromised. Therefore, these machines cannot be used as stepping stones to carry out further attacks against third parties. On the other hand, high-interaction honeypots offer real services to the attackers to interact with which makes them more risky than low-interaction honeypots. As a matter of fact, they offer a more suitable environment to collect information on attackers activities once they manage to get the control of a target machine and try to progress in the intrusion process to get additional privileges. It is noteworthy that recently, hybrid honeypots combining the advantages of low and high-interaction honeypots have been also proposed [7][8]. Both types of honeypots are investigated in the CADHo project to collect information about the malicious activities on the Internet and to build models that can be used to characterize attackers behaviors and to support the definition and the validation of the fault assumptions considered in the design of secure and intrusion tolerant systems.

During the first stage of the project, the partner of the CADHo project have focused on the deployment of a data collection environment (called Leurré.com [9]) based on low-interaction honeypots. As of today, around 40 honeypot platforms have been deployed at various sites from academia and industry in almost 30 different countries over the five continents. Each platform emulates three computers running Linux RedHat, Windows 98 and Windows NT, respectively, and various services such as `ftp`, `http`, etc. The data gathered by each platform are securely uploaded to a centralized database with the complete content, including payload of all packets sent to or from these honeypots, and additional information to facilitate its analysis, such as the IP geographical localization of packets' source addresses, the OS of the attacking machine, the local time of the source, etc.

Several analysis carried out on the data collected so far have revealed that very interesting observations and conclusions can be derived with respect to

the attack activities observed on the Internet [5][10][11][12][6]. Nevertheless, with such honeypots, hackers can only scan ports and send requests to fake servers without ever succeeding in taking control over them. The second stage of the CADHo project is aimed at setting up and deploying high-interaction honeypots to allow us to analyze and model the behavior of malicious attackers once they have managed to compromise and get access to a new host, under strict control and monitoring. The main interest is oriented in observing the progress of real attack processes and the activities carried out by the attackers in a controlled environment.

In this section, is described the preliminary lessons learned from the development and deployment of such a honeypot.

The main contributions of this section are threefold. First, the confirmation of the findings discussed in [12] showing that different sets of compromised machines are used to carry out the various stages of planned attacks. Second, it is outlined the fact that, despite this apparent sophistication, the actors behind those actions do not seem to be extremely skillful, to say the least. Last, the geographical location of the machines involved in the last step of the attacks as well as the link with some phishing activities shed a geopolitical and socio-economical light on the results of the analysis.

### 2.3.2 Related work

The most obvious approach for setting up a high-interaction honeypot consists in the use of a physical machine and to dedicate it to record and monitor attackers activities. The installation of this machine is as easy as a normal machine. Nevertheless, probes must be added to store the activities. Operating in the kernel is by far the most frequent manner to do it. Sebek[13] and Uberlogger[14] operate in that way by using Linux Kernel Module (LKM) on Linux. More precisely, they launch a homemade module to intercept interesting system calls in order to capture the activities of attackers. Data collected in the kernel are stored on a server through the network. Communications with the server are hidden on all installed honeypots.

Another approach consists in using a virtual operating system [15]. User Mode Linux (UML) is a Linux compiled kernel which can be executed as other programs on a Linux operating system. Thanks to this technique, it is possible to have several virtual operating systems running together on a single machine. Thanks to probes, activities on virtual operating systems are logged into the machine. In [16], an architecture of a honeypot using UML is presented. Uberlogger[14] can also be implemented in such an environment. VMware is also a tool used for emulation, but it emulates a whole machine instead of an operating system. Various operating systems (Windows, Gnu/Linux, etc) can be installed and executed together. It can also be used to deploy honeypots [17].

The problem with virtual honeypots is the possibility for the intruder to

detect the presence of this virtual operating system [18]. Some well-known methods, available on Internet, allow the intruder to fingerprint VMware for example. Some solutions have been developed to hide the presence of VMware (see e.g. [19]).

Compared to honeypot-solutions based on physical machines, virtual honeypots provide a cost effective and flexible solution that is well suited for running experiments to observe attacks. In particular, the number of emulated systems and their configuration can be easily changed if needed.

### 2.3.3 Architecture of the honeypot

In the implementation, is used the VMware software and to install virtual operating system upon VMware. The objective is to setup a high-interaction honeypot that can be easily configured and upgraded for different experimental studies. In particular, the intention is to emulate in the initial setup a limited number of machines, and then increase the number of emulated machines at a later stage of the project to have a more realistic target for attack that is representative of a real network.

As explained in the previous subsection, VMware workstation software [20] allows multiple operating systems to be run simultaneously on a single real host. With virtual operating systems, the cloning, the reconfiguration and the modification of the operating system is very simple. Furthermore, if the attacker succeeds in destroying some part of the operating system he has broken into, the recovery procedure is simplified compared to the case of a real operating system.

In the following, there is the description of:

1. the configuration of the honeypot;
2. the mechanism of the data capture;
3. the memory organization.

**Configuration** The objective of the experiment is to analyze the behavior of the attackers who succeed in breaking into a machine (a virtual host in the experiment). The vulnerability that he exploits is not as crucial as the activity he carries out once he has broken into the host. That's why they chose to use a simple vulnerability: weak passwords for `ssh` user accounts. In this way, the honeypot is not particularly hardened but this is intentional for two reasons. First, the interest in analyzing the behavior of the attackers even when, once logged in, they exploit a buffer overflow and become root. So, if they use some kernel patch such as Pax [21] for instance, the system will be more secure but it will be impossible to observe some behavior. Secondly, if the system is too hardened, the intruders may suspect something abnormal and then give up.

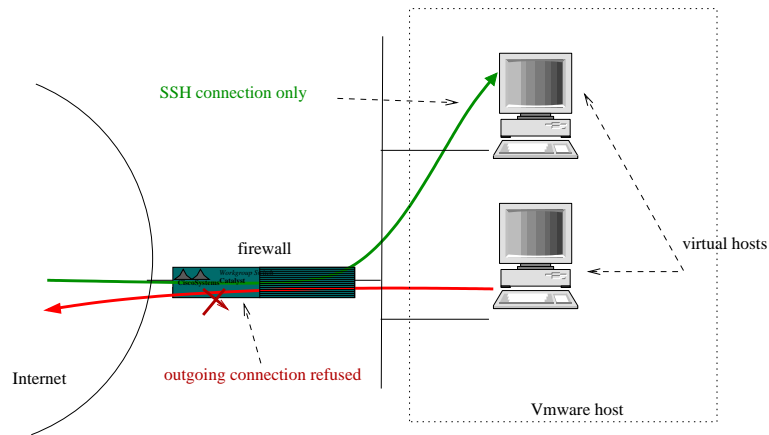


Figure 2.1: Topology of the honeypot

In the setup, only `ssh` connections to the virtual host are authorized so that the attacker can exploit this vulnerability. A firewall blocks all connection attempts, but those to port 22 (`ssh`), from the Internet (see Figure 2.1).

In order to prevent that intruders attack remote machines from the honeypot, a firewall blocks any connection from the virtual host to the outside. This does not prevent the intruder from downloading code, because he can use the `ssh` connection for that.<sup>1</sup>

This being said, as discussed later on, the lack of connectivity to the rest of the world by means of another protocol than `ssh` may look suspicious to a malicious user. They discuss the influence of this design choice in the subsection devoted to the analysis of the results of the experiments. It is showed that, instead of being a nuisance, it helps us discriminating between the various types of malicious users.

The honeypot is a standard Gnu/Linux installation, with kernel 2.6, with the usual binary tools (compiler, usual commands, etc). No additional software was installed except the `http apache` server. This kernel was modified as explained in the next paragraph. The real host is of course never used by regular users. The real operating system executing VMware is also a Gnu/Linux distribution with kernel 2.6.

**Attackers activity logging** The first objective is to log the activity of the intruders (the commands they use once they have broken into the honeypot) in a stealthy way. In order to log what the intruders do on the honeypot, they chose to modify some drivers functions `tty_read` and `tty_write` as

<sup>1</sup>As many intruders use outgoing `http` connections, the team has sometimes authorized `http` connections in the experiments for a short time under a strict control by checking constantly that the attackers were not trying to attack other remote hosts.

well as the `exec` system call in the Linux kernel. The modifications of the functions `tty_read` and `tty_write` enable us to intercept the activity on all the terminals and pseudo-terminals of the system. The modification of the `exec` system call enables us to record the list of the system calls used by the intruder. These functions are modified in such a way that the captured information is logged directly into a buffer of the kernel memory. This means that the activity of the attacker is logged on the kernel memory of the honeypot itself. This approach is not common: in most of the approaches studied, the information collected is directly sent to a remote host through the network. The advantage of the approach is that logging into the kernel memory is difficult to detect by the intruder (more difficult at least than detecting a network connection).

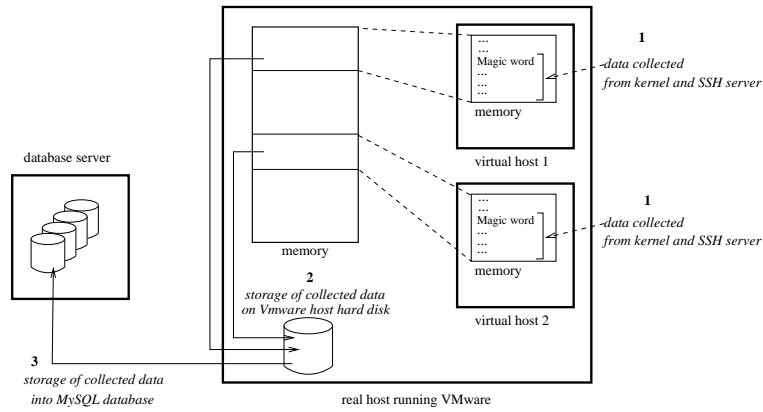


Figure 2.2: Data collection

The kernel memory of the virtual host is in fact an area of the whole memory of the real host. So, from the real host, an automatic script regularly inspects the area of the memory used by the virtual host, looking for a *magic word* indicating the beginning of the information logged into the kernel. This information is then recorded on the hard disk of the real host and then transferred to a database server (see Figure 2.2). This logging activity is executed on the real host, not on the virtual host, thus it is not easily detectable by the intruder (he cannot find anything suspicious in the list of the processes for example). Furthermore, they also decided to log this activity in such a way that it is not directly readable, even if the intruder succeeds in parsing the kernel memory of the honeypot. This activity information is compressed using the algorithm LZRW1 [22] before being logged into the kernel memory. This makes it even more difficult for an intruder to detect that he is observed.

The second objective is to record all the logins and passwords tried by the different attackers to break into the honeypot. For that purpose, they chose to add a new system call into the kernel of the virtual operating system and

they have modified the source code of the `ssh` server so that it uses this new system call. The logins and passwords are then logged in the kernel memory, in the same buffer as the information related to the commands used by the attackers. As the whole buffer is regularly stored on the hard disk of the real host, it is not needed to add other mechanisms to record these logins and passwords.

The activities of the intruder logged by the honeypot are preprocessed and then stored into an SQL database. The raw data are automatically processed to extract relevant information for further analysis, mainly:

- the IP address of the attacking machine;
- the login and the password tested;
- the date of the connection;
- the terminal associated (`tty`) to each connection;
- each command used by the attacker .

The analysis they have made are presented in the following subsections but we look first to the memory organization of the honeypot.

**Memory organization high-interaction honeypot** With the high-interaction honeypot is allowed the intruder to attack the system only at the port 22 with SSH connection (Secure Shell). What they want to do is monitoring the intruder when he is logged in for seeing what is his behavior. They modify same system calls to save the data to a kernel memory space allocated before. They do not allocate this space with `kmalloc()` because it is difficult retrieve the information managing with the virtual operating system. So they allocate at boot time same data structure. When an intruder login he utilizes same system call. He opens a `tty` so, since the `tty` is managed by the kernel like a file, the system call `open`, `read` and `write` will be called and, since it is a `tty`, `read` and `write` will call the kernel functions `tty_read` and `tty_write`. Every time the intruder launches a command there is an execution of `exec`. So they modify the `exec` system call in order to record each operation performed by the attacker. They modify the drivers `tty_read` and `tty_write` in order to catch all the information of the terminal and, if any, the code of the malware introduced by the attacker.

The structure is, as table 2.1 shows, composed by:

- **MW**: Magic Word, it is a special known word that permits to understand where the data registered in the kernel space of the virtual machine begins;
- **# op**: the number of operations performed;

- **Pointer of large buffer**: it is an integer value used to remember how many items have been recorded into the large buffer;
- **Large buffer**: it is a buffer that contains compressed data. The data are compressed because of two reasons: first, we are in kernel space and we have “only” 4 Mbytes, if we use a lot of space we could loss same monitoring attack (anyway it has never happened). Second the data are compressed it is very difficult for an intruder to understand that it is monitored.
- **Small buffer**: it is a buffer used temporary for registering the data of the attack. This data, when the small buffer is full, will be compressed and put within the large buffer.
- **Pointer of small buffer**: the same as large pointer but for the small buffer.

MW
# op
large buffer
pointer large buffer
small buffer
pointer small buffer

Table 2.1: Main structure of the honeypot memory organization

### 2.3.4 Experimental results

In this subsection, we present the results of the experiments. First is given a global statistic, in order to give an overview of the activities observed on the honeypot, then a characterization of the various intrusion processes. Finally, there is a detailed analyze of the behavior of the attackers once they manage to break into the honeypot. In this book, an *intrusion* corresponds to the activities carried out by an intruder who has succeeded to break into the system.

**Global statistics** The high-interaction honeypot has been deployed on the Internet and has been running for 131 days during which 480 IP addresses have tried to contact its `ssh` port. It is worth comparing this value to the amount of hits observed against port 22, considering all the other low-interaction honeypot platforms deployed in the rest of the world (40 platforms). In the average, each platform has received hits on port 22 from

Account	Number of connection attempts	Percentage of connection attempts	Number of pass tested
root	34251	13.77%	12027
admin	4007	1.61%	1425
test	3109	1.25%	561
user	1247	0.50%	267
guest	1128	0.45%	201
info	886	0.36%	203
mysql	870	0.35%	211
oracle	857	0.34%	226
postgres	834	0.33%	194
webmaster	728	0.29%	170

Table 2.2: `ssh` connection attempts and number of passwords tested

around approximately 100 different IPs during the same period of time. Only four platforms have been contacted by more than 300 different IP addresses on that part and only one was hit by more visitors than the high interaction honeypot deployed from the team. Even better, the low-interaction platform maintained in the same subnet as the high-interaction experimented only 298 visits, i.e. less than two thirds of what the high-interaction did see. This very simple and first observation confirms the fact already described in [12] that some attacks are driven by the fact that attackers know in advance, thanks to scans done by other machines, where potentially vulnerable services are running. The existence of such a service on a machine will trigger more attacks against it. This is what is observed here: the low interaction machines do not have the `ssh` service open, as opposed to the high interaction one, and, therefore get less attacked than the one where some target has been identified.

The number of `ssh` connection attempts to the honeypot have been recorded is 248717 (the scans on the `ssh` port are not considered here). This represents about 1900 connection attempts a day. Among these 248717 connection attempts, only 344 were successful. Table 2.2 represents the user accounts that were mostly tried (the top ten) as well as the total amount of different passwords that have been tested by the attackers. It is noteworthy that many user accounts corresponding to usual first names have also regularly been tested on the honeypot. The total number of accounts tested is 41530.

Before the real beginning of the experiment (approximately one and a half month), a machine with a `ssh` server correctly configured is deployed, offering no weak account and password. The advantage of this observation period to determine which accounts were mostly tried by automated scripts. Using this acquired knowledge, 17 user accounts have been created and they

User Account	Duration between creation and first successful connection	Duration between first successful connection and first intrusion
UA1	1 day	4 days
UA2	half a day	4 minutes
UA3	15 days	1 day
UA4	5 days	10 days
UA5	5 days	null
UA6	1 day	4 days
UA7	5 days	8 days
UA8	1 day	9 days
UA9	1 day	12 days
UA10	3 days	2 minutes
UA11	7 days	4 days
UA12	1 day	8 days
UA13	5 days	17 days
UA14	5 days	13 days
UA15	9 days	7 days
UA16	1 day	14 days
UA17	1 day	12 days

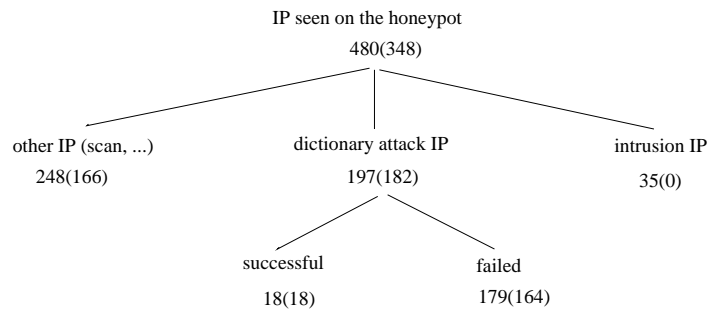
Table 2.3: History of breaking accounts

have started looking for successful intrusions. Some of the created accounts were among the most attacked ones and others not. As we already explained in the chapter, they have deliberately created user accounts with weak passwords (except for the `root` account). Then, the time between the creation of the account and the first successful connection to this account have been measured, then the duration between the first successful connection and the first real intrusion (as explained in section 2.3.4, the first successful connection is very seldom a real intrusion but rather an automatic script which tests passwords). Table 2.3 summarizes these durations (**UAi** means **User Account i**).

The second column indicates that there usually is a gap of several days between the time when a weak password is found and the time when someone logs into the system with this same password to issue some commands on the now compromised host. This is somehow a surprising fact and is described with some more details here below. The particular case of the **UA5** account is explained as follows: an intruder succeeded in breaking the **UA4** account. This intruder looked at the contents of the `/etc/passwd` file in order to see the list of user accounts for this machine. He immediately decided to try to break the **UA5** account and he was successful. Thus, for this account, the first successful connection is also the first intrusion.

**Intrusion process** In the subsection, is presented the conclusions of the analysis regarding the process to exploit the weak password vulnerability of the honeypot. The observed attack activities can be grouped into three main categories:

1. dictionary attacks;
2. interactive intrusions;
3. other activities such as scanning, etc.



X(Y) : Y IP addresses among X were also seen on the low-interaction honeypot

Figure 2.3: Classification of IP addresses seen on the honeypot

As illustrated in figure 2.3, among the 480 IP addresses that were seen on the honeypot, 197 performed dictionary attacks and 35 performed real intrusions on the honeypot (see below for details). The 248 IP addresses left were used for scanning activity or activity that are not clearly identified. Among the 197 IP addresses that made dictionary attacks, 18 succeeded in finding passwords. The others (179) did not find the passwords either because their dictionary did not include the accounts created or because the corresponding weak password had already been changed by a previous intruder. In Figure 2.3 there is also represented the corresponding number of IP addresses that were also seen on the low-interaction honeypot deployed in the context of the project in the same network (between brackets). Whereas most of the IP addresses seen on the high interaction honeypot are also observed on the low interaction honeypot, none of the 35 IPs used to really log into the machine to launch commands have ever been observed on any of the low interaction honeypots that are controlled in the whole world! This striking result is discussed here after.

**Dictionary attack** The preliminary step of the intrusion consists in dictionary attacks<sup>2</sup>. In general, it takes only a couple of days for newly created

<sup>2</sup>Let us note here that we consider as “dictionary attack” any attack that tries more than 10 different accounts and passwords.

accounts to be compromised. As shown in Figure 2.3, these attacks have been launched from 197 IP addresses. By analyzing more precisely the duration between the different `ssh` connection attempts from the same attacking machine, is evident that these dictionary attacks are executed by automatic scripts. As a matter of fact, a result of the studies is that these attacking machines try several hundreds, even several thousands of accounts in a very short time.

For the machines that succeed in finding passwords, further analysis have been made, i.e., the 18 IP addresses. By searching the database containing information about the activities of these addresses against the other low interaction honeypots four important elements of information are founded. First, none of the low interaction honeypot has an `ssh` server running, none of them replies to requests sent to port 22. These machines are thus scanning machines without any prior knowledge on their open ports. Second, the found evidences that these IPs were scanning in a simple sequential way all addresses to be found in a block of addresses. Moreover, the comparison of the fingerprints left on the low interaction honeypots highlights the fact that these machines are running tools behaving the same way, not to say the same tool. Third, these machines are only interested in port 22, they have never been seen connecting to other ports. Fourth, there is no apparent correlation as far as their geographical location is concerned: they are located all over the world.

In other words, it comes from this analysis that these IPs are used to run a well known program. The activities linked to that tool, as observed in the database thanks to all the platforms, indicate that it is unlikely to be a worm but rather an easy to use and widely spread tool.

**Interactive attack: intrusion** The second step of the attack consists in the real intrusion. They have noted that, several days after the guessing of a weak password, an interactive `ssh` connection is executed on the honeypot to issue several commands. There are reason to believe that, in those situations, a real human being, as opposed to an automated script, is connected to the machine. This is explained and justified in subsection 2.3.5. As shown in Figure 2.3, these intrusions come from 35 IP addresses never observed on any of the low-interaction honeypots.

Whereas the geographic localization of the machines performing dictionary attacks is very blur, the machines that are used by a human being for the interactive `ssh` connection are, most of the time, clearly identified. The data give a precise idea of their country, geographic address, the responsible of the corresponding domain. Surprisingly, these machines, for half of them, come from the same country, an European country not usually seen as one of the most attacking ones as reported, for instance, by the [www.leurrecom.org](http://www.leurrecom.org) web site.

Then are made more analysis in order to see if these IP addresses had tried to connect to other ports of the honeypot except for these interactive connections; and the answer is no. Furthermore, the machines that make interactive `ssh` connections on the honeypot do not make any other kind of connections on this honeypot, i.e., no scan or dictionary attack. Further analysis, using the data collected from the low-interaction honeypots deployed in the CADHo project, revealed that none of the 35 IP addresses have ever been observed on any of the platforms deployed in the world. This is interesting because it shows that these machines are totally dedicated to this kind of attack (they only targeted the high-interaction honeypot and only when they knew at least one login and password on this machine).

The conclusion for these analysis is that there are two groups of attacking machines. The first group is composed of machines that are specifically in charge of making dictionary attacks. Then the results of these dictionary attacks are published somewhere. Then, another group of machines, which has no intersection with the first group, comes to exploit the weak passwords discovered by the first group. This second group of machines is clearly geographically identified and commands are executed by a human being. A similar two steps process was already observed in the CADHo project when analyzing the data collected from the low-interaction honeypots (see [12] for more details).

### 2.3.5 Behavior of attackers

This subsection is dedicated to the analysis of the behavior of the intruders. A first characterization of the intruders, i.e. trying to know if they are humans or programs, is followed from the presentation in more details of the various actions they have carried out on the honeypot. Finally, they try to figure out what their skill level seems to be.

The analysis is concentrated on the last three months of the experiment. During this period, some intruders have visited the honeypot only once, others have visited it several times, for a total of 38 `ssh` intrusions. These intrusions were initiated from 16 IP addresses and 7 accounts were used. Table 2.4 presents the number of intrusions per account, IP addresses and passwords used for these intrusions. It is of course very difficult to be sure that all the intrusions for a same account are initiated by the same person. Nevertheless, in this deploy of the honeypot, they noted that:

- most of the time, after his first login, the attacker changes the weak password into a strong which, from there on, remains unchanged;
- when two different IP addresses access the same account (with the same password), they are very close and belong to the same country or company.

account	Number of intrusions	Number of password	Number of IP address
UA2	1	1	1
UA4	13	2	2
UA5	1	1	1
UA8	1	1	1
UA10	9	2	2
UA13	6	1	5
UA16	5	1	3
UA17	2	1	1

Table 2.4: Number of intrusions per account

These two remarks lead us to believe that there is in general only one person associated to the intrusions for a particular account.

**Type of the attackers: humans or programs** Before analyzing what intruders do when connected, the paper is oriented to identify who they are. They can be of two different natures. Either they are humans, or they are programs which reproduce simple behaviors. For all intrusions but 12, intruders have made mistakes when typing commands. Mistakes are identified when the intruder uses the backspace to erase a previously entered character. So, it is very likely that such activities are carried out by a human, rather than programs.

When an intruder did not make any mistake, they analyze how the data are transmitted from the attacker machine to the honeypot. They can note that, for `ssh` communications, data transmission between the client and the server is asynchronous. Most of the time, the `ssh` client implementation uses the function `select()` to get user input. So, when the user presses a key, this function ends and the program sends the corresponding value to the server. In the case of a copy and a paste into the terminal running the client, the `select()` function also ends, but the program sends all the values contained in the buffer used for the paste into the server. We can assume that, when the function `tty_read()` returns more than one character, these values have been sent after a copy and a paste. If all the activities during a connection are due to a copy and a paste, we can strongly assume that it is due to an automatic script. Otherwise, this is quite likely a human being who uses shortcuts from time to time (such as CTRL-V to paste commands into its `ssh` session). For 7 out of the last 12 activities without mistakes, intruders have entered several commands on a character by character basis. This, once again, seems to indicate that a human being is entering the commands. For the 5 others, their activities are not significant enough to conclude : they

have only launched a single command, like `w`, which is not long enough to highlight a copy and a paste.

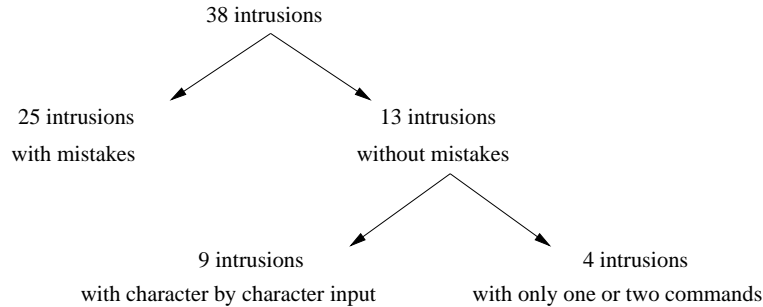


Figure 2.4: Characterization of the intrusions

**Attacker activities** The first significant remark is that all of the intruders change the password of the hacked account. The second remark is that most of them start by downloading some files. In all, but one, cases the attackers have tried to download some malware to the compromised machines. In a single case, the attacker has first tried to download an innocuous, yet large, file to the machine (the binary for a driver coming from a known web site). This is probably a simple way to assess the quality of the connectivity of the compromised host.

The command used by the intruders to download the software is `wget`. To be more precise, 21 intrusions upon 38 include the `wget` command. These 21 intrusions concern all the hacked accounts. As mentioned in subsection 2.3.3, outgoing `http` connections are forbidden by the firewall. Nevertheless, the intruders still have the possibility to download files through the `ssh` connection using `sftp` command (instead of `wget`). Thus, it is interesting to analyze the percentage of the attackers that continue their attack despite this `wget` problem. Surprisingly, they noted that only 30% of the intruders did use this `ssh` connection. 70% of the attackers were unable to download their malware due to the absence of `http` connectivity! Three explanations can be envisaged at this stage. First, they follow some simplistic cookbook and do not even know the other methods at their disposal to upload a file. Second, the machines where the malware resides do not support `sftp`. Third, the lack of `http` connectivity made the attacker suspicious and he decided to leave the system. Surprisingly enough, the first explanation seems to be the right one as observing them leaving the machine after an unsuccessful `wget` and coming back a few hours or days later, trying the same command again as if they were hoping it to work at that time. Some of them have been seen trying this several times. It comes out of this that i) they are apparently unable to understand why the command fails, ii) they are not afraid to come back to the machine despite the lack of `http` connectivity, iii) applying such

brute force attack reveals that they are not aware of any other method to upload the file.

Once they manage to download their malware thanks to **sftp**, they try to install it (by decompressing or extracting files for example). 75% of the intrusions that installed software did not install it on the hacked account but rather on standard directories such as **/tmp**, **/var/tmp** or **/dev/shm** (which are directories with write access for everybody). This makes the activity of the hacker more difficult to identify because these directories are regularly used by the operating system itself and shared by all the users.

Additionally, they have identified four main activities of the intruders. The first one is launching **ssh** scans on other networks but these scans have never tested local machines. Their idea is to use the targeted machine to scan other networks. So, for the administrator of the targeted network, it is more difficult to localize them. The program used by most intruders, which is easy to find on the Internet, is **pscan.c**.

The second type of activity consists in launching **irc** clients. Some examples are **emech** [23] and **psyBNC**. Names of binary files have regularly been changed by intruders, probably in order to dissimulate them. For example, the binary files of **emech** have been changed to **crond** or **inetd**, which are well known binary file names and processes on Unix systems. The **irc** clients are used for Denial Of Service attacks (DoS).

The third type of activity is trying to become root. Surprisingly, such attempts have been observed for 3 intrusions only. Two rootkits were used. The first one exploits two vulnerabilities: a vulnerability which concerns the Linux kernel memory management code of the **mremap** system call[24] and a vulnerability which concerns the internal kernel function used to manage process's memory heap[25]. This exploit could not succeed because the kernel version of the honeypot does not correspond to the version of the exploit. The intruder should have realized this because he checked the version of the kernel of the honeypot (**uname -a**). However, he launched this rootkit anyway and failed. The other rootkit used by intruders exploits a vulnerability in the program **ld**. Thanks to this exploit, three intruders became **root** but the buffer overflow succeeded only partially. Even if they apparently became **root**, they could not launch all desired programs (removing files for example caused access control errors).

The last activity observed in the honeypot is related to phishing activities. It is difficult to make precise conclusions because only one intruder has attempted to launch such an attack. He downloaded a forged email and tried to send it through the local **smtp** agent. But it looked like a preliminary step of the attack because the list of recipient emails was very short. It seems that it was just a preliminary test before the real deployment of the attack.

**Attackers skill** Intruders can roughly speaking be classified into two main categories. The most important one is relative to *script kiddies*. They are inexperienced *hackers* who use programs found on the Internet without really understanding how they work. The next category represents intruders who are more dangerous. They are named “black hat”. They can make serious damage on systems because they are expert in security and they know how to exploit vulnerabilities on various systems.

As already presented in 2.3.5 (use of `wget` and `sftp`), the observation is that intruders are not as clever as expected. For example, for two hacked accounts, the intruders don’t seem to really understand the Unix file access rights (it’s very obvious for example when they try to erase some files whereas they don’t have the required privileges). For these two same accounts, the intruders also try to kill the processes of other users. Many intruders do not try to delete the file containing the history of their commands or do not try to deactivate this history function (this file depends on the login shell used, it is `.bash_history` for example for the `bash`). Among the 38 intrusions, only 14 were cleaned by the intruders (11 have deactivated the history function and 3 have deleted the `.bash_history` file). This means that 24 intrusions left behind them a perfectly readable summary of their activity within the honeypot.

The IP address of the honeypot is private and the they have started another honeypot on this network. This second honeypot is not directly accessible from the outside, it is only accessible from the first honeypot. They have modified the `/etc/motd` file of the first honeypot (which is automatically printed on the screen during the login process) and added the following message: “In order to use the software XXX, please connect to A.B.C.D”. In spite of this message, only one intruder has tried to connect to the second honeypot. They could expect that an experienced hacker will try to use this information. In a more general way, they have very seldom seen an intruder looking for other active machines on the same network.

One important thing to note is relative to fingerprinting activity. No intruder has tried to check the presence of VMware software. For three hacked accounts, the intruders have read the contents of the file `/proc/cpuinfo` but that’s all. None of the methods discussed on Internet was tested to identify the presence of VMware software [18][26]. This probably means that the intruders are not experienced hackers.

### 2.3.6 Conclusion

In this section, we have presented the results of an experiment carried out over a period of 6 months and described in detail in the paper [1]. The observation is that there are various steps that lead an attacker to successfully break into a vulnerable machine and his behavior once he has managed to take control over the machine.

The findings are somehow consistent with the informal know how shared by security experts. The contributions that the paper bring is in performing an experiment and rigorous analyses that confirm some of these informal assumptions. Also, the precise analysis of the observed attacks reveals several interesting facts. First of all, the complementarity between high and low interaction honeypots is highlighted as some explanations can be found by combining information coming from both set ups. Second, it appears that most of the observed attacks against port 22 were only partially automatized and carried out by script kiddies. This is very different from what can be observed against other ports, such as 445, 139 and others, where worms have been designed to completely carry out the tasks required for the infection and propagation. Last but not least, honeypot fingerprinting does not seem to be a high priority for attackers as none of them has tried the known techniques to check if they were under observation. It is also worth mentioning a couple of important missing observations. First, they did not observe scanners detecting the presence of the open ssh port and providing this information to other machines in charge of running the dictionary attack. This is different from previous observations reported in [12]. Second, as most of the attacks follow very simple and repetitive patterns, they did not observe anything that could be used to derive sophisticated scenarios of attacks that could be analyzed by intrusion detection correlation engine. Of course, at this stage it is too early to derive definite conclusions from this observation.

Therefore, it would be interesting to keep doing this experiment over a longer period of time to see if things do change, for instance if a more efficient automation takes place. They would have to solve the problems of weak passwords being replaced by strong ones though, in order to see more people succeeding in breaking into the system. Also, it would be worth running the same experiment by opening another vulnerability into the system and verifying if the identified steps remain the same, if the types of attackers are similar. Could it be, at the contrary, that some ports are preferably chosen by script kiddies while others are reserved to some more elite attackers? This is something that they are in the process of assessing.

This chapter is the state of art at the beginning of the high-interaction honeypot data processing system. During 6 months a huge quantity of data storage is achieved and there are not tools for retrieving these information in a comfortable way. High-interaction honeypot data processing system is an interface that allow to retrieve information from the honeypot database. The feature of such a system are explained in the next chapters.

## Chapter 3

# Honeypot data processing system

A honeypot is a machine connected to a network but that no one is supposed to use. If a connection occurs, it must be, at best an accidental error or, more likely, an attempt to attack the machine.

Every new attempt is saved within a database, that contains the raw data. In the CADHo project we only consider the high-interaction honeypot. To retrieve the information from this system, the team has written scripts. These scripts are written in shell programming and the visualization is on the console. In the current version of the prototype the analysis of the attack data, which is the core of the CADHo project, is very uncomfortable. An operator must execute scripts and read the results without any possibility to have these data formatted in any other way he wants. The aim of our project is to build a wrapper that allows to retrieve these information in a more efficient and effective way. The wrapper is the high-interaction honeypot data processing system and it will have tools to synthesize data retrieved in a graphical way. From now we call “system” the high-interaction honeypot data processing system, see figure 1.1. The legacy system is the old program, composed of scripts, that allowed to retrieve the information. The legacy system will continue to work normally without any additional constraints.

In this chapter we analyze the architectural features of the system of data processing. We utilize the UML (Unified Modeling Language) in order to analyze and model the architecture; one important reason to use UML is to allow the exchange of information between the components of the group following a formalism that do not allow a lot of ambiguity. The first part of the chapter explains the general matters concerning the application, so the architecture client-server, the layers and the tiers. The section 3.2 analyze in detail the function of the application. The section 3.3 analyze the class diagram to both the client and the server side, introducing the patterns utilized.

**Technologies used** The database of the legacy system is implemented in MySQL technology so the SQL requests will be performed utilizing this technology. The language of implementation is java because it is suitable implementing GUI, but also because of the experience of the group and the large documentation available on the net.

### 3.1 Architecture

The system has an architecture client-server, so it is a distributed application accessible from the web. The interface allow to retrieve and manipulate the raw data. It's required the possibility of automatizing the data obtained. In fact the client that will perform any operation on the system could be a user or a program.

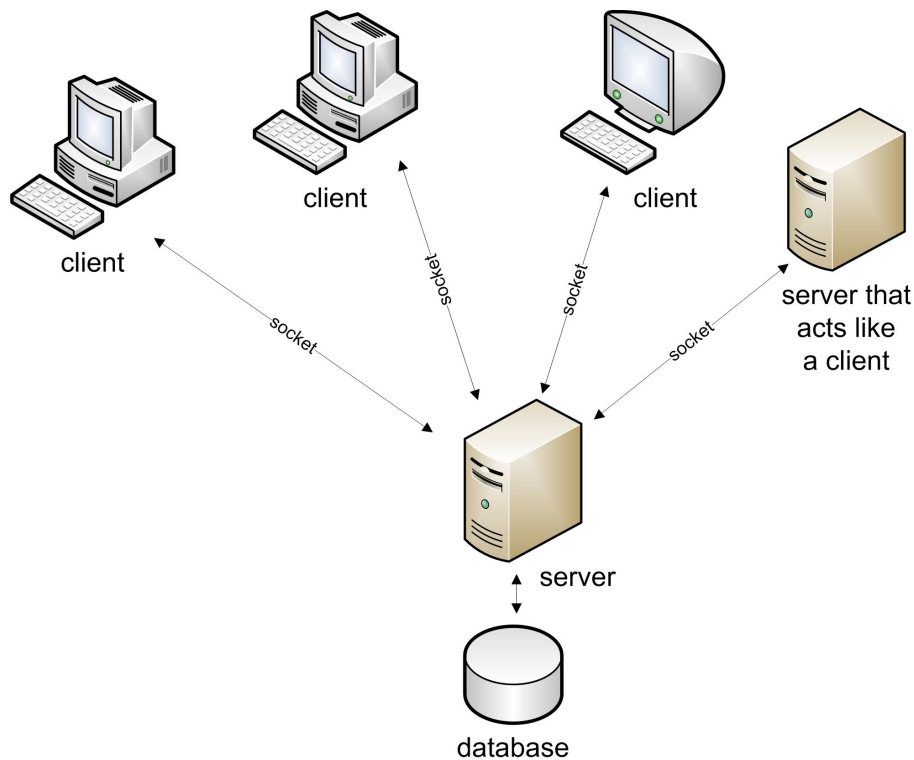


Figure 3.1: Client server architecture

We use the definition of “client” of the book [27]: the client is any user or program that wants to perform an operation over the system. Clients interact with the system through a presentation layer. Automating the processing of the data recorded on the honeypot means that these data could be utilized from another server in an automatic way. There will be a communication protocol, see chapter 5, that says to the server, that acts like a client, how

to retrieve the services of our application, figure 3.1. One requisite of the new system is that it should be portable and platform independent. What's more, at the state of the art, we need a good scalability because the number of users could enough increase during the future years, with scalability we allow a good performance in this case.

We leave open the possibility to have another server that interacts with our server but at the moment we will have three possibilities for interacting with the server. The first possibility is that the server will be accessed from

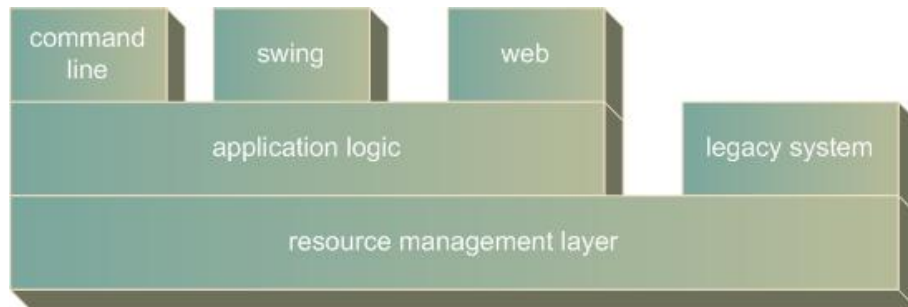


Figure 3.2: The three layers

the web, so from a browser equipped with the necessary plugins, that allow to download and execute the application. The second possibility is to access to the server installing a java Swing application on the host. The Swing application is a java interface, with the same look and feel of the application downloaded from the web, that allows to process the requests. The third possibility is to process the request from a terminal connection, like telnet for example, and visualize the raw data on the screen. If you know exactly the details of the communication protocol you can anyway understand, with some difficulties, what the data means.

With the first two cases we obtain the data in a suitable form. The third case is admitted because same browser do not dispose of the right plugins for the java application and, moreover, a telnet connection is very light. With telnet you need just a tiny client to perform a request instead of a fat client for the first two cases.

The architecture of the system is three layers<sup>1</sup> and two tiers; the three layers are visualized in figure 3.2. The application logic<sup>2</sup> is shared between the client and the server, what it contains is explained later in the text. The application logic interfaces with the resource management<sup>3</sup> with a JDBC

<sup>1</sup>There are many advantages to use a three layer architecture, for more information see [27].

<sup>2</sup>The application logic determines what the system actually does. It takes care of enforcing the business rules and establish the business processes, for more information see [27].

<sup>3</sup>The resource manager deal with the organization (storage, indexing and retrieval) of

driver. Our architecture is two tiers because the application logic at the server is physically in the same place of the database. Instead, as already said, the client could be remote.

The place of installation of the database will be probably at the city of Nice, France, and people will access the information utilizing our system.

### 3.2 Use Case specification

We begin in this section to model the concept using UML. The formalisms used in this section are the same of the books [27] and [28]. The application will develop two use cases: the Dictionary attack and the Terminal information. In the future more use cases will be added to these implemented here.

We leave the name of the actor like “user” because the client of the application could be very generic. The details on the meaning of the functionalities will be explained in the next chapter.

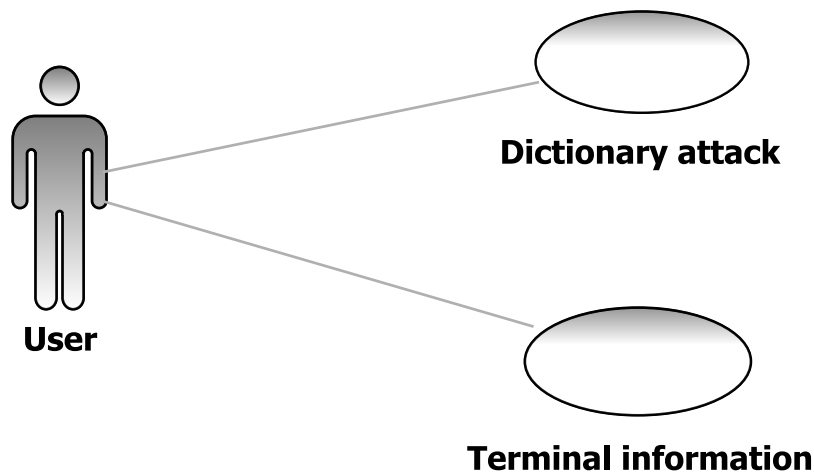


Figure 3.3: Use case diagram.

---

the data necessary to support the application logic. This is typically a database but it can also be a text retrieval system or any other data management system providing querying capabilities and persistence, for more information see [27].

Name	Dictionary attack
Initiator	User
Goal	Retrieve the information about all attacks performed in a requested time distinguishing two attacks from an attack elapsed time

### Main success scenario

1. User asks to retrieve dictionary attack information on the main menu;
2. User inserts the instants(date and hour) of the begin and the end of the time window. He inserts also the interval time to be considered for two attacks.
3. The system provides all the information required in two different visualizations: the first one is a table and the second one is a graphic view of the table.

### Extensions

3. Any error occurs
  - a)The system shows an error message and returns to main menu.

Name	Terminal information
Initiator	User
Goal	Retrieve the terminal information about all attacks performed in a requested time. Retrieve the information about all attacks

### Main success scenario

1. User asks to retrieve terminal information on the main menu;
2. User inserts the instants(date and hour) of the begin and the end of the time window;
3. The system provides a table containing all the information: the terminal name, the begin of the attack and the end of the attack, one row for each attack.
4. The user chooses one attack;

5. The system displays all the activities recorded by that terminal at the moment of the attack.

### Extensions

- 3a The system provides an empty table

- a) The user returns to insert the dates.

- 3b. Any error occur

- a)The system shows an error message and return to main menu.

5. Any error occur

- a)The system shows an error message and return to main menu.

## 3.3 Class diagram

### 3.3.1 General

As we can notice analyzing the use cases and the scenario's problem, our system will interrogate a remote database respect to the client and it will provide same services. We will call these services functionalities. One functionality is a particular request that we want to ask to the server. These functionalities have same common features and same peculiar features of the service that they represent. Moreover we know that the service will be used by many people and so it is important to provide an user-friendly graphical interface.

### 3.3.2 Server side

**Server side organization** The server class contains the main program of the server side application. Inside the `main()` program we create all the functionalities: dictionary attack, terminal information, ecc.... Then the `main()` waits any connection on a specified port.

The server must be multithreaded because many requests could arrive consecutively and, any time, a new thread is launched to avoid the blocking of the request for the other clients. So the class `ServerSocketThread` extends the class `thread` overriding the method `run`.

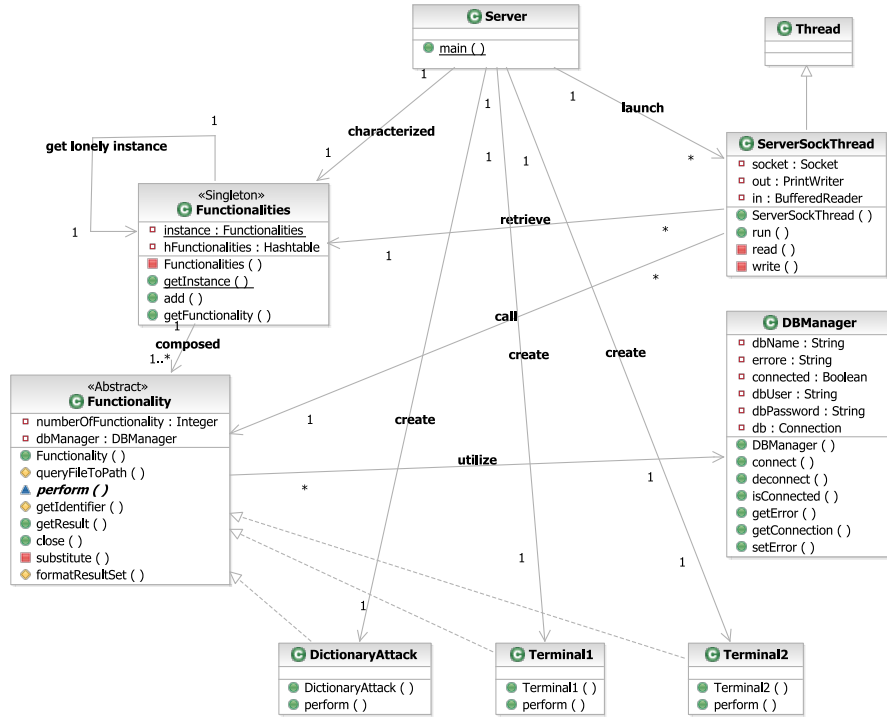


Figure 3.4: Server class diagram

For each request the server establishes a socket connection<sup>4</sup> with the client. The server operations are wrapped on the class `ServerSocketThread`: the method `read` and `write` are used respectively for the socket operations read and write.

`DBManager` is the wrapper class for all the database operations. Essentially it is used for connecting and unconnecting the database and to manage its generated errors. This class contains attributes of login information, `dbPassword` and `dbUser`, and the identification of the database into the attribute `dbName`.

The `Functionalities` class is a Singleton pattern and it is used to keep all the information offered by the system. It should be a Singleton pattern because we want just one instance of the class, we can access to the instance with the method `getInstance()` that returns the lonely object. The class allows to add other functionalities through the method `add()` where we specify the functionality we want to add.

With the `getFunctionality()` method we obtain the object `Functionality` through the specification of the functionality number<sup>4</sup>.

The hash table contains all the pairs: functionality key, functionality

<sup>4</sup>See chapter 5 Protocol communication.

instance. Functionality key is an integer number that identifies in an univocal way the functionality<sup>5</sup>, functionality instance is the object of the class *Functionality*.

Functionality is an abstract class that represents the common aspects of the entity functionality. The various functionalities, dictionary attack, terminal information, etc. . . , should implement the abstract method *perform()* that contains all the peculiarity of the functionality. For example, all the test on the type of exchanged data between the client and the server are peculiar of the functionality.

The attribute *numberOfFunctionality* is the integer that identifies the functionality in an univocal way.

The *getIdentifier()* method is used to obtain the functionality identification.

The attribute *dbManager* keeps the connection with the database to allow the execution of the SQL query, performed from the method *getResult()*.

The *queryFilePath()* method is used to obtain the path file that contains the SQL query. All the file paths that contain the SQL query are contained in the file *c:/query\_file.txt*<sup>5</sup>.

The *close()* method is used to close all the resources opened to perform the query, the database connection mainly.

The *substitute()* method, substitutes the SQL query parameters with the right values send from the client<sup>5</sup>.

The *formatResultSet()* method, formats the result obtained from the execution of the SQL script<sup>6</sup>.

In same functionalities we need to perform a double client-server interaction, this is the case of the terminal information functionality. In these cases we firstly obtain same information and, in function of them, we perform, to the client side, the second interaction to obtain the result. When a new request will arrive to the server, it will open a socket. What does it will do if the functionality is composed of a double interaction? We may manage this situation in two ways. The server may leave open the socket and it may wait that the client performs the second request. Contrarily the server may close the socket and open another socket later when the second request will arrive and threatening it as a new functionality. What the server can do is to break the functionality in two sub-functionality.

This second organization is possible because of our interactions are stateless for the server side. This means that the new demand depends only on the data it is carrying on and on the state that the client has reached. The state is independent of the state of the server. So there is not a great utility to leave an opened session to wait the second request of the client. Contrarily the disadvantage is evident: the overloading of the server that becomes less scal-

---

<sup>5</sup>See chapter 6, Query SQL.

<sup>6</sup>The result is formatted as specified in chapter 5, Protocol communication.

able. So we choose to manage one functionality as two sub-functionalities. In practice these are treated like two different functionalities. For these reasons the functionalities as terminal information are implemented with two classes: Terminal1 and Terminal2, the number emphasizes the interaction's number. Moreover it is for all these reasons that the cardinality between the classes Functionality and ServerSocketThread is (1,n). So the socket is opened for each simple functionality<sup>7</sup> and for each sub-functionality<sup>8</sup>. This organization guarantees better performance and is more easy to implement because the sub-functionality assumes more independence with respect to the bigger functionality.

**Server side advantages** All the functionalities inherit from the class Functionality and this has same of advantages. To the server side, in fact, we have to perform the SQL script relative to the functionality and check that all the data exchanged between the entities are conformed to that specified in the protocol communication. There is a huge quantity of common features to all the functionalities; the class Functionality is born with the aim of gathering all the common features. Each real implementation of one functionality inherits from the class Functionality and develops only the part that is dependent on itself. One of the aims of the project is to keep the system in the way we can easy upgrade it later. In fact, it is anticipated that a lot of functionalities will be added in the future. With our server side organization the evolution of the system becomes very easy, because there is a lot of code that will not be rewritten. As you can notice from the class diagram at page 38, the various functionalities implement only the method of the abstract class from which they inherit.

Another advantage is the use of the hash tables. The class Functionalities keeps the class that associates the key of the functionality with the object functionality. This allows to access the functionality object in a faster way, just the time of accessing the hash table, and when the number of the functionalities becomes greater it avoids huge switch-cases into the code.

The other hash table contains all the data sent from the client and it is used to separate the net level from the rest of the application. The parameters may be retrieved in a comfortable way, passed to the other methods all together and put in the hash table already tested.

### 3.3.3 Client side

**MVC Model-view-controller** Model-view-controller (MVC) is an architectural pattern used in software engineering. In complex computer appli-

---

<sup>7</sup>We call simple functionality the one only functionality.

<sup>8</sup>We call sub-functionality one interaction of one functionality composed from more interactions. In the rest of the book the word functionality will be utilized with the meaning of simple functionality or sub-functionality.

cations that present lots of data to the user, one often wishes to separate data (model) and user interface (view) concerns, so that changes to the user interface do not impact the data handling, and that the data can be reorganized without changing the user interface. The model-view-controller solves this problem by decoupling data access and business logic from data presentation and user interaction, by introducing an intermediate component: the controller.

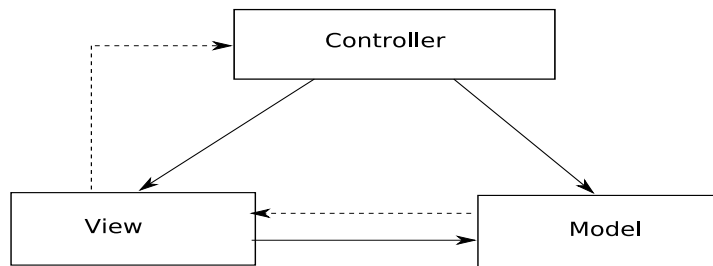


Figure 3.5: A simple diagram depicting the relationship between the Model, View, and Controller. Note: the solid lines indicate a direct association, and the dashed line indicate an indirect association (e.g.: observer pattern).

It is common to split an application into separate layers: presentation (UI), domain, and data access. In MVC the presentation layer is further separated into View and Controller.

- **Model** The domain-specific representation of the information on which the application operates. It is a common misconception that the model is another name for the domain layer. Domain logic adds meaning to raw data (e.g., calculating if today is the user's birthday, or the totals, taxes and shipping charges for shopping cart items). Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the data access layer because it is understood to be underneath or encapsulated by the Model.
- **View** Renders the model into a form suitable for interaction, typically a user interface element.
- **Controller** Processes and responds to events, typically user actions, and may invoke changes on the model.

MVC is often seen in web applications, where the view is the actual HTML page and the code which gathers dynamic data and generates the

content within the HTML is the controller. Finally the model is represented by the actual content, usually stored in a database or XML-files.

Though MVC comes in different flavors, control flow generally works as follows:

1. The user interacts with the user interface in some way (e.g., user presses a button);
2. A controller handles the input event from the user interface, often via a registered handler or callback;
3. The controller accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller updates user's shopping cart);
4. A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view. (However, the observer pattern can be used to allow the model to indirectly notify interested parties, potentially including views, of a change.)
5. The user interface waits for further user interactions, which begins the cycle anew.

**Event channel** In big architectures it is very important to uncouple the different part of the application, keep them independent in order to not cause a lot of code changing if there will be same changes or increasing of the software volume in the future. The solution is often to have a central object that knows the other objects and, in general, nobody knows each other. All the objects exchange the messages with the only object known, the message is then forwarded to the interested object. All the objects are listening on the channel that forwards the events, for this reason the channel object will be called EventChannel. When the listeners recognize the events that are addressed to them, they catch and process them. The event channel is the core of the client application but at the same time is only a point where everybody should pass, it does not add important things to the execution of the functionality.

The EventChannel has three methods for each functionality "func": addFunc(), removeFunc() and fireFunc().

- addFunc() is needed to register a listener that is going to receive information on that functionality "func";
- removeFunc() is needed to remove a listener previously registered;

- `fireFunc()` is needed to perform a broadcast to all the objects interested in that specific functionality “func”.

Each listener is registered to the EventChannel in function of the functionality, so there is a set of listeners for each functionality<sup>9</sup>.

**New custom event definition** The information are exchanged utilizing custom events that are functionality dependent. For each functionality, in fact, the information to be exchanged are different. What we do is inherit from an event that contains the common information and extend this object with the peculiar characteristic of the functionality.

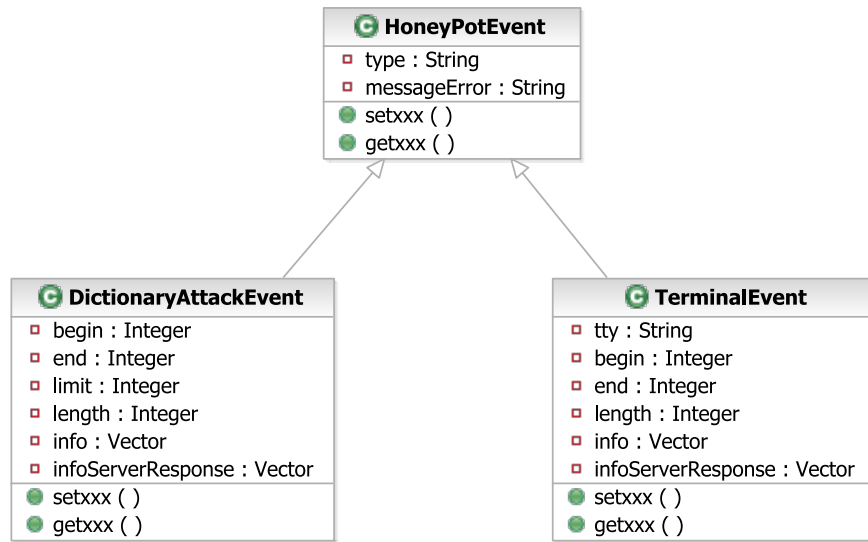


Figure 3.6: Class event diagram

In the figure we define the new event `HoneyPotEvent`<sup>10</sup>. This event contains the attributes `type` and `messageError` that are common to all the functionalities. The `type` is the information about which is the next step the system should perform. This field has the aim to help an object to understand if the next one to perform a computation is him or is not. `messageError` contains the information about the error in the case same have occurred. The new events contain also all the get and set in order to manipulate the attributes, for sake of conciseness, they are marked with xxx instead of the name of all the attributes.

For each functionality we define a new event that inherits from `HoneyPotEvent`. As you can notice from the figure 3.6, the other information

<sup>9</sup>For information about the implementation of the EventChannel see the annex.

<sup>10</sup>See the annex for information about the implementation of the creation of custom events.

carried from the events are functionality dependent.

Later in the text there are the sections that explain the details of the various functionalities and from which the new attributes of the event derive.

**Client side model** We have introduced the concept of MVC pattern and event channel, now we introduce a portion of the client side architecture identifying the different parts in function of the patterns viewed. We now show a portion of the architecture, we do not show all the classes here. The figure 3.7 refers to one simple functionality<sup>11</sup>.

We have adopted the MVC pattern customizing it for our needs. The direct link of figure 3.5 between the model and the view is been deleted with the aim of integrating also the event channel concept. Moreover we have added a dashed line between the model and the controller. All the messages are dispatched from the event channel, so also the model must go through the controller.

We explain the architecture of figure 3.7, these are the various steps:

1. The user selects the functionality (in the picture we have just one functionality);
2. The event channel understands which functionality it should call and it wakes up, with a broadcast communication, all the observers that are listening. The common case is that just one object will wake up but there are cases were we have more objects. So, in our example, there is only one waked up object and it receives the event.
3. The object, waked up from the event channel, processes the event showing the associated view that is an interface for the user;
4. The user sets the parameters and generates the Swing event<sup>12</sup>. This step is in dashed line because the generation of the event is automatic in Java.
5. The controller catches the event of the Swing and creates a custom event that is loaded with all the information in order to perform the request. The event is sent to the event channel;
6. Like the step 2;
7. The Process object executes the request through the net and the server. The result is loaded on the custom event and sent to the event channel object.
8. Like the step 2;

---

<sup>11</sup>Remember that simple functionality means single interaction.

<sup>12</sup>For example the event of button pressed or same other widget.

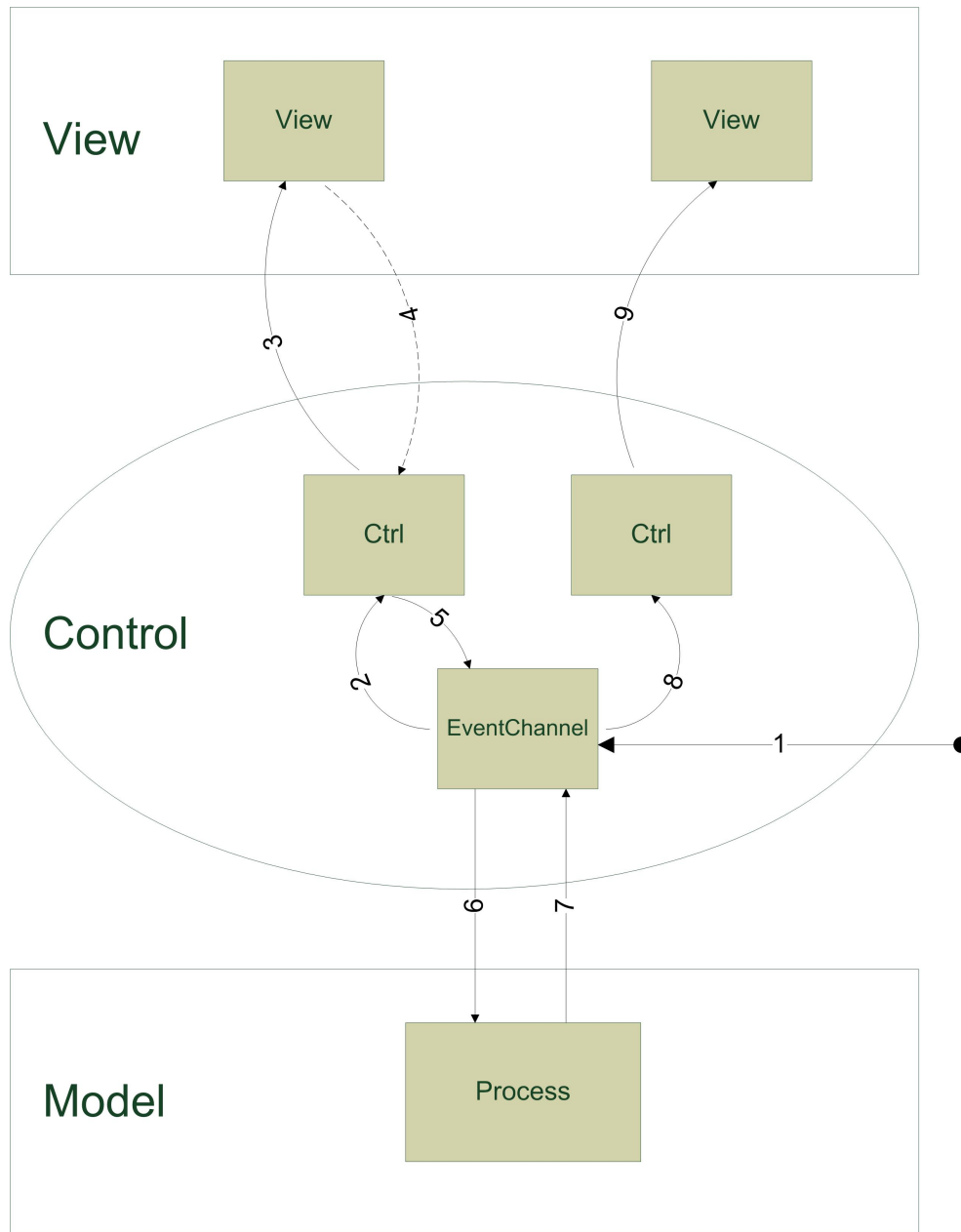


Figure 3.7: MVC and event channel matched on a portion of the real system

9. The controller understands that the process is addressed to him and processes the event. This means that it displays the obtained data into an interface.

In the case we have a double interaction functionality we have to add another cycle from the View of the step 9, to the server and then come back to a View with a new interface. We do not use the same classes but we create others. In the main diagram there are all the functionalities and some of all are with double interaction. See the figure 3.8.

As you notice from the picture 3.7 we have utilized both the concepts of event channel and MVC pattern. The last one, in particular, has a little bit changed. The View portion is strictly the one of graphical interface. The portion of Control is composed of two objects: the Ctrls that are controllers of the graphical interface and the event channel that is an universal controller that works like a dispatcher. The Model portion is interfaced only with the Controller.

In the picture 3.7 the schema is simplified. In the real class diagram it is composed of more classes as explained in the next paragraph.

**Advantage of the client side model** By adopting this client model organization we have kept independent the various services of the application. To add a new functionality we repeat the schema of figure 3.7 and the lonely thing shared between the functionalities is the event channel, so the mix of code is minimum.

Also inside each functionality we leave a lot of flexibility. First of all we benefit of all the advantages of the MVC<sup>13</sup> that are to keep separate the data (Model), from the visualization (View), through a new layer (Controller). Moreover we have uncoupled the various parts that perform one functionality. To perform one simple functionality we have to follow these steps: obtain the user data, process and display the result. These three steps are extremely independent in our application, the various objects do not have to know each other. In more complex functionalities what we have to do is add two new parts completely independent from the other objects.

**Client side organization** The real structure of the class diagram is a little bit different from the one explained in the previous paragraphs.

The classes are disposed in order to underly the different layers of the MVC pattern. At the top we have all the Views that are marked with the stereotype Swing. At the centre we have the Controller layer countersigned with the name of the classes that terminate with Ctrl, and the EventChannel. At the bottom of the figure there is the Model layer composed of the classes Process, Thread and ClientSock.

---

<sup>13</sup>See the MVC pattern paragraph for more details on the advantages of MVC.

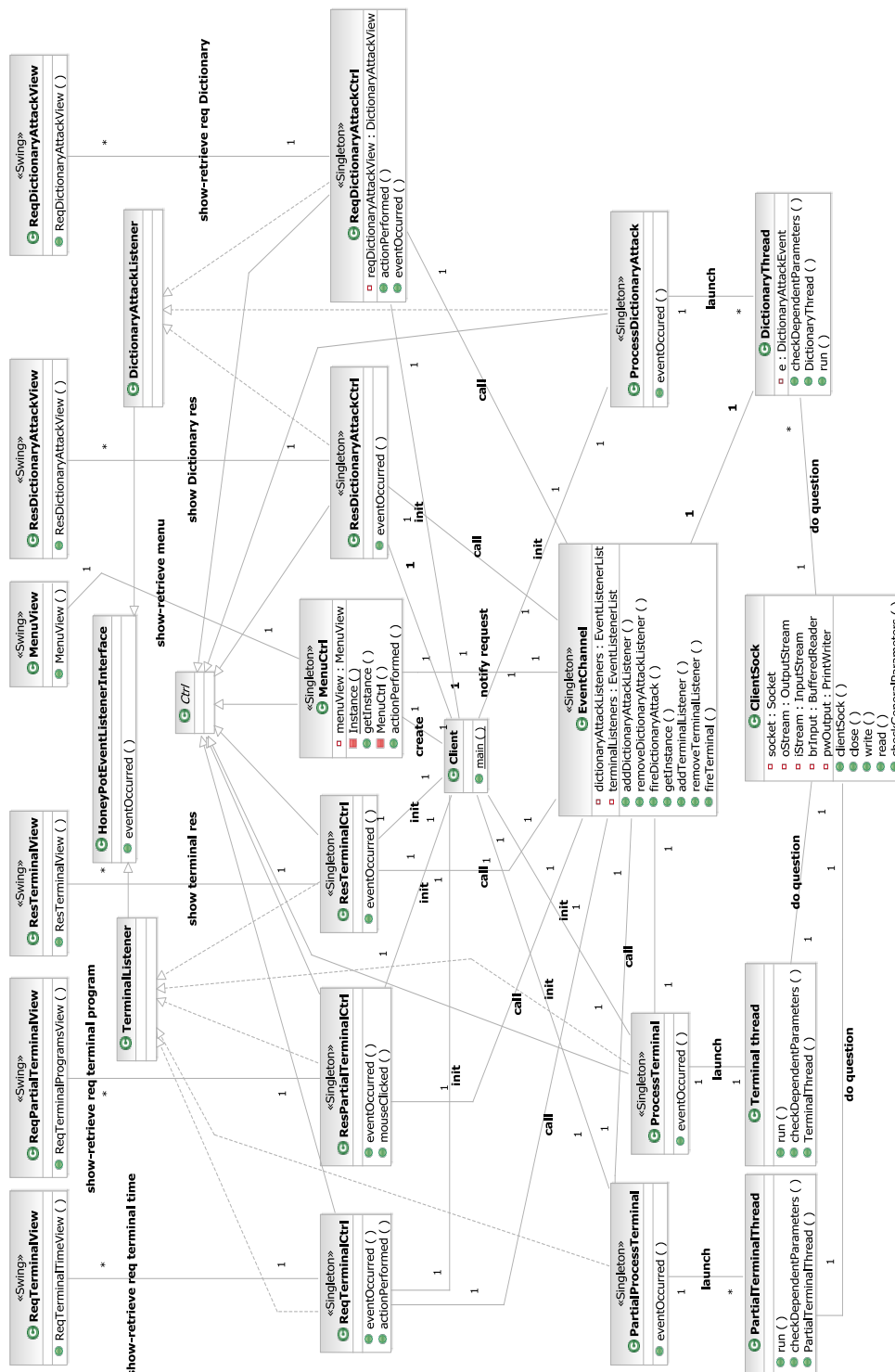


Figure 3.8: Complete client class diagram

As you can notice the classes of the View layer have only the method to be created. These classes have only the goal of displaying the user interface.

In the Control layer, all the control classes inherit from a class called Ctrl. We want to model that all the classes Ctrl are controllers of the graphical interfaces of the View layer. The events of the View layer are captured from the respective class of control and then the request is forwarded to the EventChannel. Inside the methods *actionPerformed()* we do the catch of the events generated from the widgets of the form set from the user. It is useful to capture the events this way in order to uncouple completely the View from the Control.

The Ctrls of one functionality inherit also from the class XxxListener. The Xxx is the name of the different functionalities. Moreover the class XxxListener generalize the interface HoneyPotEventListener, with the method *eventOccurred()*. This is the common method of all the listeners and the method that the EventChannel utilizes to notify that an event of interest has occurred. This interface represents that all the Ctrls are listeners on the event dispatched from the EventChannel.

If we analyze, for example, the case of Dictionary attack functionality, the Ctrls are ReqDictionaryAttackCtrl and ResDictionaryAttackCtrl, these inherit from the class Ctrl and implement the method *eventOccurred()* of the class DictionaryAttackListener. The class DictionaryAttackListener does not has peculiar methods of this functionality, anyway we leave this potentiality on the general scheme.

In the Model layer we have to add something respect to the model of figure 3.7. This application manages non trivial query SQL and net operations with the socket. These computations could demand many seconds to be performed. We want to give the possibility to perform a second request while the server has not finished the processing of the first one. So it is not enough to have the lonely server multithread, we need also a client that allows multithreading computations. It is for this reason that on the diagram we have class XxxThread, where Xxx represents the name of the functionality they implement. Following the previous example of Dictionary attack the thread is called DictionaryThread.

The set of classes Process are used to process the request addressed to the server. They launch a thread that retrieves the data from the server. The Process object loads the data retrieved on a custom event and it sends them to the EventChannel. As you can notice the Process objects are controllers too, they inherit from the classes Ctrl and Listener. Anyway they have a different meaning turned to the data, it is for this reason that they are placed on the Model layer. The lonely method that they have to implement is *eventOccurred()* that is needed, like the other controllers, to interface with the EventChannel.

The thread classes implement the method *run()* inherited from the extension of the class Thread that belongs to the standard API. This is the method

called to launch a new thread. It also implements the method *checkDependentParameters()* that is used to check the parameters retrieved from the net. Here we check only the parameters that are functionality dependent<sup>14</sup>.

The exchange of parameters with the net is demanded to the class ClientSock that is called from the thread to read and write on the socket through the method *read()* and *write()*. This class gathers also a general method useful to all the threads off all the functionalities. The method is *checkGeneralParameters()* and it is used to check the general parameters retrieved from the net. Here we check the general parameters, the parameters that are common to all the functionalities<sup>14</sup>.

All the classes that have a control function realize a Singleton pattern, these are marked with the stereotype Singleton. We want a single instance of these classes. The registration of this single instance on the EventChannel occur through the class Client.

The Client class contains the *main()* program that is the application that launches the main menu of all the functionalities available. The class MenuCtrl and MenuView are the classes that manage the menu, respectively the Control and the View. It is from the menu View that the user chooses the functionality to run.

---

<sup>14</sup>For more details see the chapter 5 Protocol communication.

## Chapter 4

# Functionalities

In this chapter we explain the meaning of the two functionalities developed for the high-interaction data processing system: the dictionary attack and the terminal information. In both cases we first explain the general meaning in term of computer science and security, and then we show the interface developed.

### 4.1 Dictionary attack

#### 4.1.1 What is a dictionary attack?

In cryptanalysis and computer security, a dictionary attack is a technique for defeating a cipher or authentication mechanism by trying to determine its decryption key or passphrase by searching a large number of possibilities.

In contrast with a brute force attack, where all possibilities are searched through exhaustively, a dictionary attack only tries possibilities which are most likely to succeed, typically derived from a list of words in a dictionary. Generally, dictionary attacks succeed because most people have a tendency to choose passwords which are easy to remember, and typically choose words taken from their native language.

Dictionary attacks may be applied in two main situations:

- in cryptanalysis, in trying to determine the decryption key for a given piece of ciphertext;
- in computer security, in trying to circumvent an authentication mechanism for accessing a computer system by guessing passwords.

In the latter case, the effectiveness of a dictionary attack can be greatly reduced by limiting the number of authentication attempts that can be performed each minute, and even blocking further attempts after a threshold of

failed authentication attempts is reached. Generally, 3 attempts is considered sufficient to cope with mistakes made by legitimate users; beyond that, one can safely assume that the user is a malicious attacker.

In our case we don't want to increase the security of our system because the honeypot wants to attract a large number of attacks.

There is some commonality between these situations. For instance, an eavesdropper may record a challenge-response authentication exchange between two parties and use a dictionary attack to try to determine what the password was. Or, an attacker may be able to obtain a copy of the list of encrypted passwords from a remote system; assuming the users are mostly English speakers, the attacker could attempt to guess the passwords at their leisure, by encrypting each of a list of English words and comparing each encryption against the stored encrypted version of users' passwords. Since users often choose easily guessed passwords, this has historically succeeded about 4 times out of 10 when a reasonably large list is used. Dictionaries for most human languages (even those no longer used) are easily accessible on the Internet, meaning even the use of foreign words is practically useless in preventing dictionary attacks.

An example of a dictionary attack occurred in the Second World War, when British codebreakers working on German Enigma-ciphered messages used the German word *eins* as part of the attack; *eins*, the word for the number one, appeared in 90 percent of all ciphertexts, as the Enigma machine's keyboard had no numerals <sup>1</sup>.

#### 4.1.2 Dictionary attack functionality

During the last years, the project CADHo has accumulated a large amount of dictionary attack data, addressed to our honeypots. These data have been stored on a big central database. The dictionary attack functionality is used to retrieve and visualize these information in order to analyze them.

Our aim is to retrieve the dictionary attack information, occurred in a given temporal interval specified by the user, through the definition of the begin and end parameters. One dictionary attack is associated to one IP address.

Nevertheless there is also another issue, due to the fact that most of the IP addresses of the Internet are dynamically assigned. The main consequence is that IP addresses that appear more than once in the database may not refer to the same machine. Besides, let us suppose that a machine performs two dictionary attacks, executed within 12 hours from each other. What this means, does this second attack belong to the first one or is it a new one?

One dictionary attack is composed of one or several consecutive login attempts. Two consecutive attempts belongs to the same dictionary attack

---

<sup>1</sup>Some might classify this as a known plaintext attack.

if the time interval separating these attempts is less then a specified limit, see figure 4.1. For these reasons, besides the temporal interval another pa-

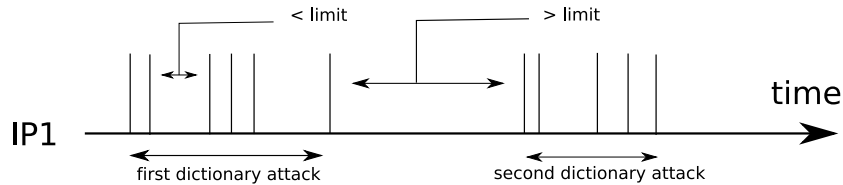


Figure 4.1: The parameter limit

parameter must be established that points out the maximum time between two dictionary attacks. For more information and an example see chapters: 5 and 6 and especially the example 6.3.

So the user must specify three numbers: begin, end and limit. The interface to specify these numbers is figure 4.2

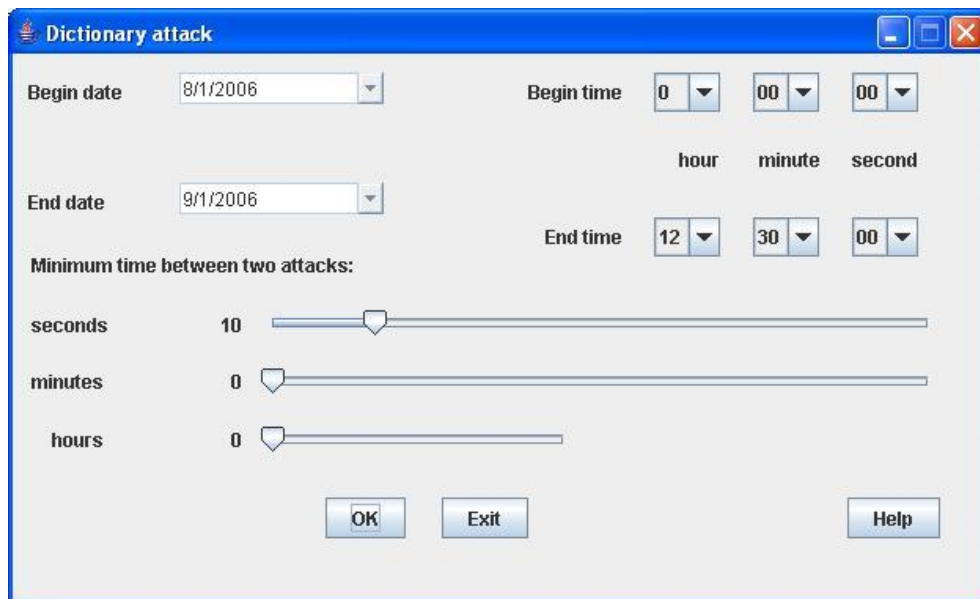


Figure 4.2: Main window of dictionary attack

On the top of the interface we have the widgets to specify the begin instant. This is composed from a calendar, that allow to specify the date, and three boxes in order to specify hour, minute and second of the day. On the middle we have the end instant specification working the same. On the bottom, we set the maximum duration parameter (limit) with three slidebar in order to specificate hours, minutes and seconds. The maximum

value of limit may be one day. In the figure we have setted the fields of the interface with the following data:

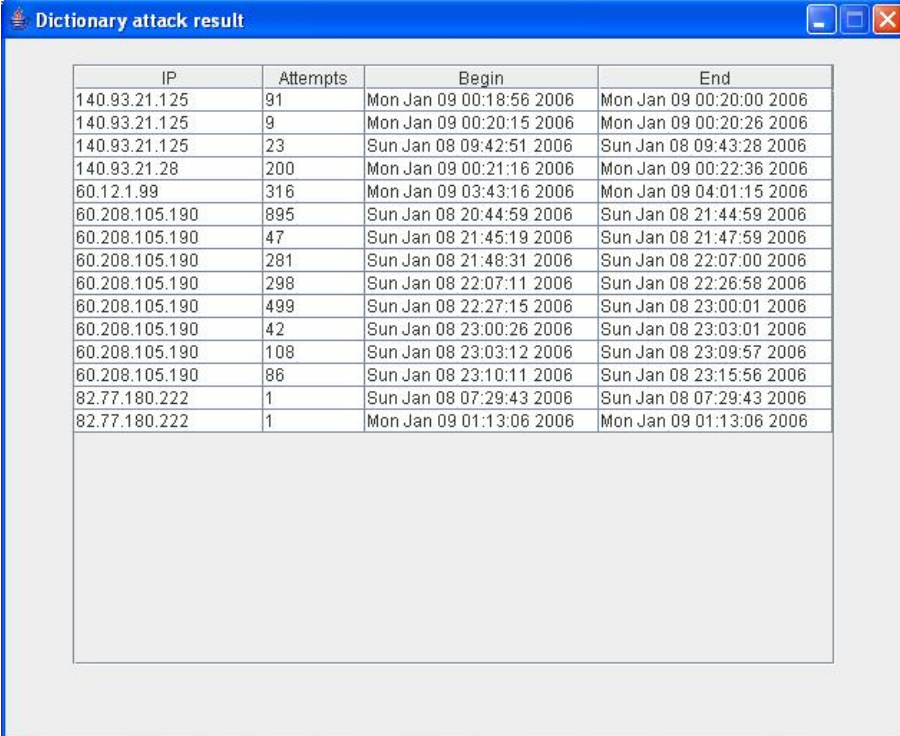
- begin = date: 8/1/2006 hour: 0:00:00;
- end = date: 9/1/2006 hour: 12:30:00;
- limit = 10 seconds.

From now we will call begin and end these two parameters.

When an OK click occurs we get two representations of the same data: a table and a graph.

### 4.1.3 Table representation of the data

On the table, we get a line for every dictionary attack. On the line we obtain the IP address of the machine, the number of attempts that have been tried, the begin and the end of the attacks, figure 4.3 (the parameters used in order to generate the table are the same of the figure 4.2).



IP	Attempts	Begin	End
140.93.21.125	91	Mon Jan 09 00:18:56 2006	Mon Jan 09 00:20:00 2006
140.93.21.125	9	Mon Jan 09 00:20:15 2006	Mon Jan 09 00:20:26 2006
140.93.21.125	23	Sun Jan 08 09:42:51 2006	Sun Jan 08 09:43:28 2006
140.93.21.28	200	Mon Jan 09 00:21:16 2006	Mon Jan 09 00:22:36 2006
60.12.1.99	316	Mon Jan 09 03:43:16 2006	Mon Jan 09 04:01:15 2006
60.208.105.190	895	Sun Jan 08 20:44:59 2006	Sun Jan 08 21:44:59 2006
60.208.105.190	47	Sun Jan 08 21:45:19 2006	Sun Jan 08 21:47:59 2006
60.208.105.190	281	Sun Jan 08 21:48:31 2006	Sun Jan 08 22:07:00 2006
60.208.105.190	298	Sun Jan 08 22:07:11 2006	Sun Jan 08 22:26:58 2006
60.208.105.190	499	Sun Jan 08 22:27:15 2006	Sun Jan 08 23:00:01 2006
60.208.105.190	42	Sun Jan 08 23:00:26 2006	Sun Jan 08 23:03:01 2006
60.208.105.190	108	Sun Jan 08 23:03:12 2006	Sun Jan 08 23:09:57 2006
60.208.105.190	86	Sun Jan 08 23:10:11 2006	Sun Jan 08 23:15:56 2006
82.77.180.222	1	Sun Jan 08 07:29:43 2006	Sun Jan 08 07:29:43 2006
82.77.180.222	1	Mon Jan 09 01:13:06 2006	Mon Jan 09 01:13:06 2006

Figure 4.3: Table result of the functionality dictionary attack

As you can notice, there are more lines with the same IP address. This is due to the parameter limit that is not an infinite number.

#### 4.1.4 Graphic representation of the data

Our aim is to obtain a first glance idea of the table data; we have developed a graphic interface. We use an UNIX system representation of the time. In the UNIX systems the time is counted in milliseconds beginning from a determined instant of the history. This instant is what is called “zero epoch” and it corresponds to the *January 1, 1970*.

In the interface a Cartesian coordinate system is visualized with the IP addresses on the axis y and the time on the axis x, figure 4.4. On the axis x we represent the seconds therefore, any date is in relationship 1 to 1 with a number that represents the seconds beginning from the epoch. We consider

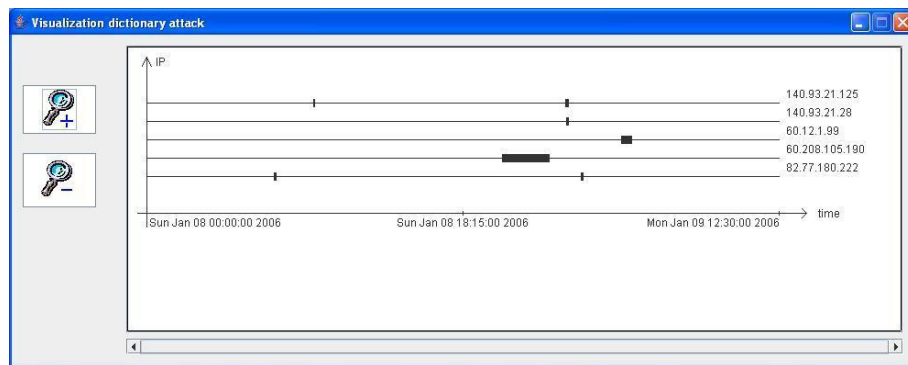


Figure 4.4: Graphic visualization of dictionary result

as viewport of our interface the space of plan among the axes. There is a line for each IP address present in the dictionary attack result and the graph changes dynamically in comparison to it. The temporal window considered is given by the user in the preceding step with the parameters begin and end of the form, as shown in figure 4.2 . We can interact with the graph making some zooms forward or backward and moving the scrollbar.

In the first visualization the scrollbar cannot be moved and begin/end of the graph exactly coincides with the parameters begin/end specified by the user. We can notice the 5 IP addresses instead of a lot of rows visualized on the table. In fact the attacks visualized on the table are reassumed by the graphic but the first time we have a small granularity. We can increase the granularity through the zoom.

The various attacks from the IP address 60.208.105.190 are drawn, in this first image, as single long attack; taking a great granularity the attacks can be visualized in their real form so it will be separated. Making a positive zoom we change the parameters, begin/end of the graph will coincide with some new parameters calculated according to a formula.

To produce the graphical representation we make a relation between the seconds and the pixels of the screen, we need to consider the size of the screen. The number of the pixels is got in a dynamic way so we will not

have problems of visualization in the case the size of the screen changes in future installations of the software on devices of different dimensions. At any instant of the time *pointSec* is mapped on a pixel *x* of the screen if this temporal instant reenters in the window of time considered. We call *totalSec* the total number of seconds, from the parameters *begin* and *end* specified by the user, we get *x* according to the proportion:

$$totalSec : pointSec = totalPixels : x$$

from which we obtain:

$$x = \frac{totalPixels \cdot pointSec}{totalSec}$$

Our temporal window is relative to the parameters *begin/end*. The parameter *pointSec* will be therefore an instant related to the parameter *begin*,  $pointSec = instantSec - begin$ , where *instantSec* is an instant of the history in seconds. Also the number *x* of pixels to be drawn is related to the axis *y* that has coordinates  $x = 0$ , so we have to set the pixel *x* beginning from the pixel of the axis *y*.

If we make a positive zoom, the figure is magnified only on the axis *x* and the bars are lengthened according to the new parameters. The parameter *totalSec* is changed according to the formula:

$$totalSec = \frac{(end - begin)}{zoom}$$

At the first instance, *zoom* is equal to one and when we perform same zoom it will be multiplied for a fixed value, for example two. Making a positive zoom there is a loss of information, there are some points that remain out of the viewport. These information can be all visualized through the scrollbar. A parameter “move” tells us that we are moved by the absolute center <sup>2</sup> of our time. Every time that the scrollbar is moved the parameter *move* is modified in function of the movement of the scrollbar. Accordingly we make the refresh of the paint including and/or excluding some attacks from the viewport. Once calculated the *x* through the proportion, that is function of the parameter already modified *totalSec*, a check is made in order to see if the point should be drawn or not. If the calculated pixel is out of our range of values, it is not drawn. Our range of values is from 0 to *totalPixels*.

The result of the zoom that we get continuing the precedent example is the figure 4.5. You notice the change of the temporal references *begin/end* and the size of the bars that represent the attacks. Also the scrollbar has changed, now we can interact with it. One attack, on the last IP address, is lost by the viewport but it can be visualized moving the picture with the scrollbar. Performing same zooms and moving the scrollbar we can see that the large bar splits in more different bars showing that it is composed of multiple attacks, see figure 4.6.

---

<sup>2</sup>The absolute center is given by  $\frac{(end - begin)}{2}$ .

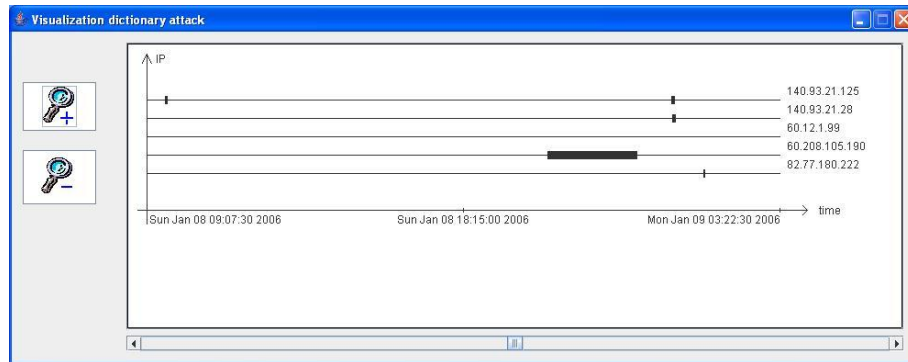


Figure 4.5: Zoom visualization

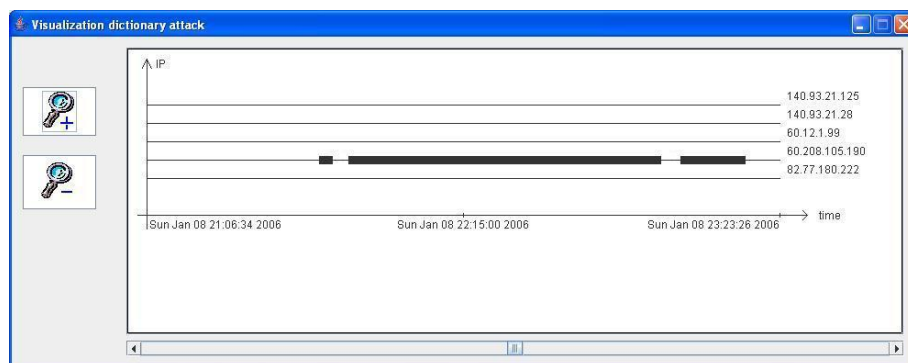


Figure 4.6: Split visualization of the attacks

## 4.2 Terminal information

### 4.2.1 What is a terminal?

**Historically** A computer terminal is an electronic or electromechanical hardware device that is used for entering data into, and displaying data from, a computer or a computing system. Typically it provides a text terminal interface over a serial line.

Early user terminals connected to computers were generally electromechanical teleprinters (TTYs: abbreviation of teletypewriters). However these were too slow for most production uses. By the early 1970s, many in the computer industry realized that an affordable video data entry terminal could supplant the ubiquitous punch cards and permit new uses for computers that would be more interactive.

Early video computer displays were sometimes nicknamed “Glass TTYs” and used individual logic gates, with no CPU. One of the motivations for development of the microprocessor was to simplify and reduce the electronics required in a terminal. Most terminals were connected to mainframe computers and often had a green or amber screen. Typically terminals communicate with the computer via a serial line, often using the RS-232 serial interface.

Later, so called intelligent terminals were introduced, such as the VT52 and VT100 made by DEC, both of which are still widely emulated in software. These were called “intelligent” because they had the capability of interpreting escape sequences to position the cursor and control the display. The VT100 terminal implemented the sophisticated ANSI standard for functions such as controlling cursor movement, character set, and display enhancements.

Today, most PC telnet clients provide emulation of the most common terminal, the DEC VT100 for example which has become the de facto standard used by terminal emulators.

**Contemporary** Since the advent and subsequent popularization of the personal computer, few genuine hardware terminals are used to interface with computers today. Using the monitor and keyboard, modern operating systems like Linux and the BSD derivatives feature virtual consoles, which are mostly independent from the hardware used. Graphical user interface (or GUI) like the X Window System, one’s display is typically occupied by a collection of windows associated with various applications, rather than a single stream of text associated with a single process. In this case, one may use a terminal emulator application within the windowing environment. This arrangement permits terminal-like interaction with the computer (for running a command line interpreter, for example) without the need for a physical terminal device.

The ubiquitous Unix terminal window is used for both local and remote access; where the connection goes is not the business of the terminal emulator itself, it just communicates through a pseudo terminal interface. Many different terminal emulators are available for the X Window System, like `xterm` for example.

A terminal emulator, terminal application, `term`, or `tty` for short, is a program that emulates a “dumb” video terminal within some other display architecture. Though typically synonymous with a command line shell or text terminal, the term terminal covers all remote terminals, including graphical interfaces. A terminal emulator inside a graphical user interface is often called a terminal window.

A terminal window allows the user access to text terminal and all its applications such as Command-Line Interfaces (CLI) and text user interface applications. These may be running either on the same machine or on a different one via `telnet`, `ssh`, or `dial-up`.

On Unix-like operating systems it is common to have one or more terminal windows connected to the local machine. Terminals usually support a set of escape sequences for controlling color, cursor position, etc.

**Pseudo terminal implementation** For each pseudo terminal, the operating system kernel provides two character devices: a master device and a slave device.

The slave and master devices, in their most common deployment, form an association between a Unix shell and a terminal emulation program or some sort of network server. The slave device file has the appearance and supported system calls of any text terminal. Thus it has the understanding of a login session and session leader process (which is typically the shell program).

The master device file is the endpoint for communication with the terminal emulator. It receives the control requests and information from the other party over this interface and responds accordingly.

**Pseudo terminal applications** Important applications of pseudo terminals include `xterm` and similar terminal emulators in the X Window System and other window systems (such as the Terminal application in Mac OS X), in which the terminal emulator process is associated with the master device and the shell is associated with the slave. Any terminal operations performed by the shell in a terminal emulator session are received and handled by the terminal emulator process itself (such as terminal resizing or terminal resets). The terminal emulator process receives input from the keyboard and mouse using windowing events, and is thus able to transmit these characters to the shell, giving the shell the appearance of the terminal emulator being an underlying hardware object.

Other important applications include remote login handlers such as `ssh` and `telnet` servers, which serve as the master for a corresponding shell, bridged by a pseudo terminal.

#### 4.2.2 Terminal information functionality

Our honeypots need a login to be accessed. Our studies are targeted to attackers that perform the attack from UNIX like systems. Thus the violation of the system is always done with a pseudo terminal slave, in other words with a shell, with a SSH (Secure Shell) connection. The login is climb over by the dictionary attack and the attack is performed once the attacker has entered into the system.

This functionality is aimed to obtain the information of what the attacker do from the terminal. The terminal is always a `pts` (pseudo terminal slave) and the aim of the functionality is to print all the activities typed from the attackers. The schema of an attack is as follows:

- the attacker opens a terminal, a shell;
- the attacker tries to login to a remote server, our honeypot for example, performing a dictionary attack;
- if the attacker logins successfully then he will perform some operations, he will execute some programs.

Our aim is to reproduce exactly what the attacker writes on the terminal. When the machine of the attacker and our honeypot are connected the honeypot records a lot of information that are stored to the main database.

Our application shows the information in a suitable way in order to analyze the data. The steps of the interaction between the user and the application are explained in section 3.2.

The first interface is a little bit simpler than the one of dictionary attack. On the dictionary attack we have the parameter `limit` that has no sense in the terminal information functionality. The rest of the parameters are the same of dictionary attack. To perform this request to the application two parameters are necessary: the begin and the end of the time interval. The interface to specify these numbers is shown in figure 4.7.

On the top of the interface we have the widgets to specify the begin instant. This is composed from a calendar, that allows to specify the date, and three boxes in order to specify hour, minute and second of the day. On the middle we have the end instant specification working the same. In the figure we have set the fields of the interface with the following data:

- begin = date: 1/1/2006 hour: 0:00:00;
- end = date: 4/3/2006 hour: 0:00:00;

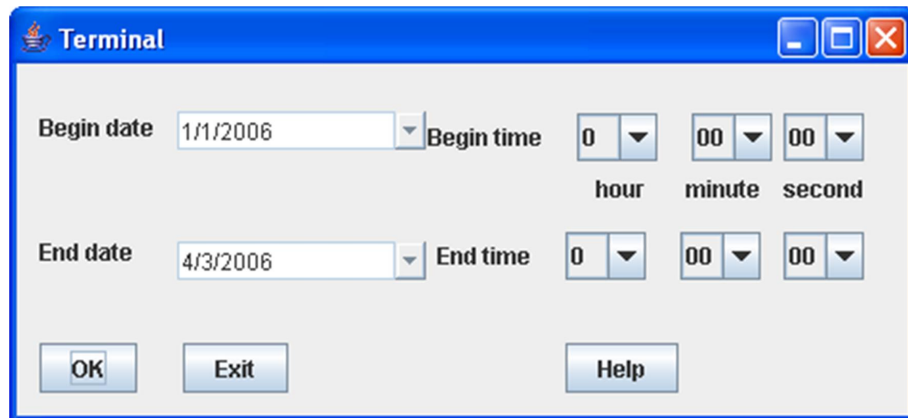


Figure 4.7: Main window of terminal information

This functionality is composed of two interactions<sup>3</sup>. The first interaction obtains all the terminals that have performed an attack during the specified period of time, i.e., the begin and the end interval. In the example we want to identify all the terminals between the first January 2006 and the four March 2006.

When an OK click occurs we get a table of three columns: the tty name, the begin and the end of the session opened by that tty. This is showed in figure 4.8. In the figure we visualize all the terminal names that the attackers have utilized to perform the attack and moreover the time references of the session opened in order to connect to our honeypot. If we click on one of the displayed lines we start the second interaction. So clicking on one of the tty of the table we ask to retrieve all the activities the attacker has typed while that session. One example result is the figure 4.9.

The command “w” prints summaries of system usage, currently logged-in users, and what those users are doing. So the attacker is maybe looking if he is the only user of the system.

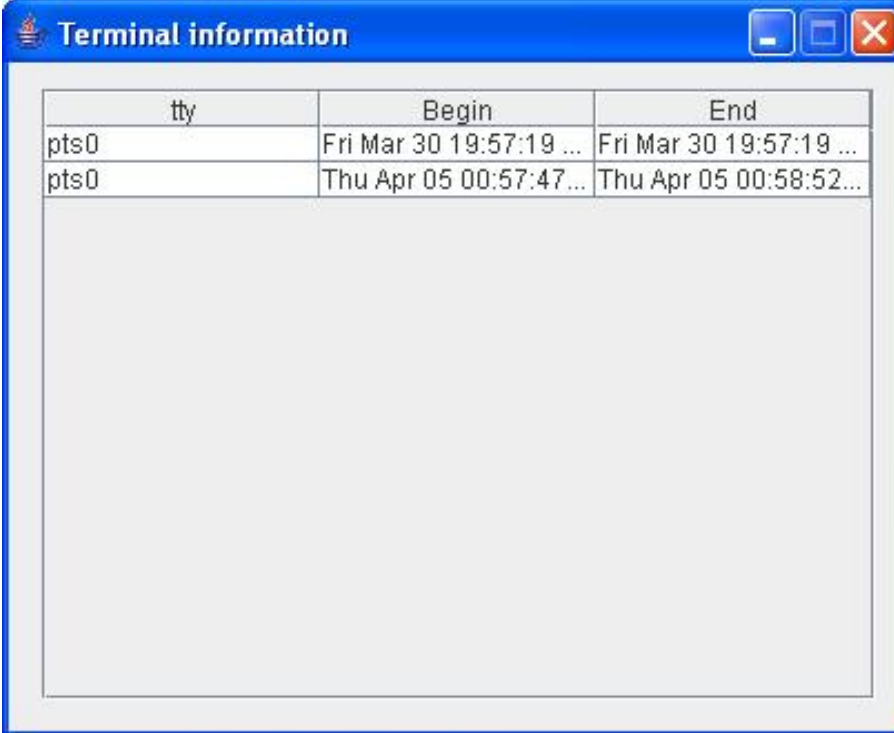
With the command “wget www.poison.lydo.org/t3.tgz” the attacker is trying to download the file t3.tgz but this operation fails.

Then he displays some information about the host with the command “cat /proc/cpuinfo”.

Describing the attacker activities is out of the scope of the project of mine, so we do not give a lot off explications concerning these results.

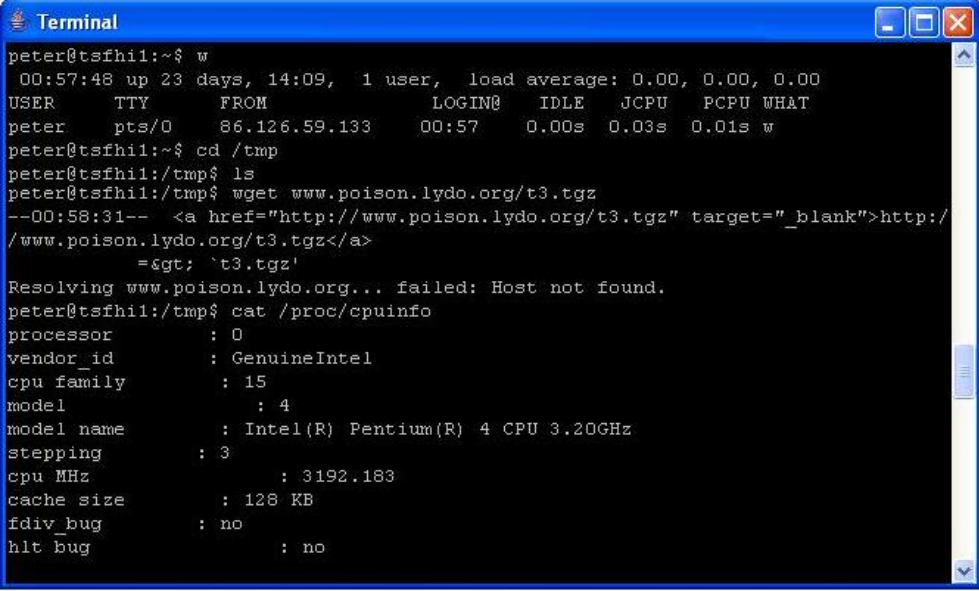
---

<sup>3</sup>See the chapter 5, Protocol communication, for more information.



tty	Begin	End
pts0	Fri Mar 30 19:57:19 ...	Fri Mar 30 19:57:19 ...
pts0	Thu Apr 05 00:57:47...	Thu Apr 05 00:58:52...

Figure 4.8: First interaction window of terminal information



```

peter@tsfh11:~$ w
 00:57:48 up 23 days, 14:09,  1 user,  load average: 0.00, 0.00, 0.00
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
peter     pts/0    86.126.59.133    00:57    0.00s  0.03s  0.01s  w
peter@tsfh11:~$ cd /tmp
peter@tsfh11:/tmp$ ls
peter@tsfh11:/tmp$ wget www.poison.lydo.org/t3.tgz
--00:58:31--  <a href="http://www.poison.lydo.org/t3.tgz" target="_blank">http://
/www.poison.lydo.org/t3.tgz</a>
              => `t3.tgz'
Resolving www.poison.lydo.org... failed: Host not found.
peter@tsfh11:/tmp$ cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 15
model          : 4
model name     : Intel(R) Pentium(R) 4 CPU 3.20GHz
stepping       : 3
cpu MHz        : 3192.183
cache size     : 128 KB
fddiv_bug      : no
hlt_bug        : no

```

Figure 4.9: Second interaction window of terminal information

## Chapter 5

# Protocol communication

In this chapter we talk about the protocol utilized on both client and server side to communicate. The first part is a general consideration on the advantages or disadvantages of the different technologies that we can use. Then we talk about the details of the protocol developed and implemented.

### 5.1 General

What we need is a communication channel between the server and the client. There are a lot of technologies that allow a channel between the two parts. The programming language that we use is Java, so the first idea to develop the channel could be utilizing the power of this middleware, for example, the Remote Method Invocation(RMI). This technology is powerful and comfortable to use. Unfortunately there are same disadvantages. Writing the bridge between the client and the server in this way means that both the net interface should be written in Java. We are binded to the platform while we would like that the two parts are completely independent.

One solution is to utilize socket that is a system independent technology. This means that we can implement the server side in a language that is different from the one of the client side. If we do not have the platform independent problem we could create an Object that contains all the information to be transferred to the other side. Utilizing the class Serializable of Java we transform the Object in an equivalent String that can be converted another time to the beginning Object. The sender transforms the object into a String, the String is sent through the net and then the receiver transforms again the String in Object. This way we have transferred the data in a very easy and comfortable way because we have utilized the already implemented API. In the conversion the data are “encrypted” behind the Java organization; this means that this solution is platform dependent.

It is for this reason that we cannot use class like Serializable, so we created a customized protocol to transfer the data. What we do is build

a string that can be read from any kind of implementation. The string is formatted in a fixed way in order to manage the retrieval of the information at the other side of the channel. Moreover we add to the pure information same special characters. Here we explain the communication protocol that we are proposed and implemented.

We define a number “#”, at the begin of the string, which identifies the functionality that the client request or maybe the interaction step number within a functionality. This parameter is a char, so 255 functionalities are allowed.

#	
---	--

Table 5.1: Functionality’s number

The functionalities are as follows:

1. Dictionary attack;
2. Terminal first interaction;
3. Terminal second interaction;

For the terminal functionality we have two interactions so we need two numbers to manage this situation. We use a char token to separate the different fields of the protocol. The token is “:” like the implementation of other utility of Unix systems. The field between the token are the information

#	:		:		:	
---	---	--	---	--	---	--

Table 5.2: Token’s structure

needed to perform the query of the client. Each functionality has a protocol to exchange the information between client and server. It means that all the fields are specified for the request and for the answer.

The parameters are inserted in the packets with an identification for each of them. This identification will allow to match same information needed to the other side (i.e.: the parameters of the SQL parameter query). The parameter is separated from the identification with an “=”. So we put for each parameter the pair “PARAMETER=parameter”.

**Request** The request message is composed of a header and same parameters. The header is composed from: REQUEST, LENGTH. The number of functionality is the REQUEST. The LENGTH header is an integer number, it counts the number of the parameters from the first parameter, so the header is not included. The LENGTH parameter is useful for reasons of testing and for future enhancement of the system.

REQUEST=#	:	LENGTH=length	:	PARAMETER=p1	:	...
-----------	---	---------------	---	--------------	---	-----

Table 5.3: Request: headers and parameters

**Answer** The answer is composed of the raw data retrieved and a header. The header is the ANSWER number, the LENGTH and the TYPE of the answer. The ANSWER parameter is important in order to be sure that the answer is relative to that request. This is the number of functionality as the request. The LENGTH parameter is the length of the data beginning from the first data. The string TYPE parameter is important in order to distinguish a normal answer from an error answer. If same errors have occurred the type parameter will be “error”. If the answer is correct the type parameter will be “result”.

In the case the answer is empty the data string returned will be empty, there will be just the type parameter. In the case same error occur to the

ANSWER=#	:	LENGTH=length	:	TYPE=result	:	d1	:	d2
----------	---	---------------	---	-------------	---	----	---	----

Table 5.4: Answer: headers and parameters

server side, the string returned will be: We don’t care about the field length.

ANSWER=#	:	LENGTH=x	:	TYPE=error	:	message error
----------	---	----------	---	------------	---	---------------

Table 5.5: Answer: error headers and parameters

Message error is a string that describe the error.

Note that the order of the parameters at request time is not important, instead it’s important for the answer.

For sake of performance, for each interaction the server is unconnected from the client and it reconnects after, for the second interaction. This way we will avoid that the server has a lot of session opened for “thinking time” at the client side. The data at the client side are obtained to be studied, so in general this could mean that one session could be open for a long time. This way we avoid useless server employment.

## 5.2 Dictionary attack

**Request** The number ‘1’ is reserved to perform this functionality. Moreover we have other three fields:

- b: is the begin of the time interval to be considered;
- e: is the end of the time interval to be considered;

- l: is the limit time for distinguishing two different attacks.

REQUEST	:	LENGTH	:	BEGIN	:	END	:	LIMIT
=1		=3		=b		=e		=l
				int		int		int

Table 5.6: Request dictionary attack

An example is: BEGIN=1136073600, End=1167631200, LIMIT=5, the begin corresponds to 01/01/2006 and the end corresponds to 01/01/2007.

**Answer** The answer is composed of four fields that are repeated for all the instances of attacks founded:

- IP: is the IP address of the attacker;
- Attempts: is the number of attempts of that attack;
- Begin: is the begin time of the attack;
- End: is the end time of the attack.

ANSWER	:	LENGTH	:	TYPE	:	attempts	:	begin	:	end
=1		=num		=result						
						int		int		int

Table 5.7: Regular answer dictionary attack

### 5.3 Terminal information

The Terminal information functionality has a multiple information interchange between the server and the client. There are two messages for each part. The number '2' and '3' are reserved for this functionality.

**First request** The number '2' is reserved to perform this first request. Moreover we have two other fields:

- b: is the begin session time to be considered;
- e: is the end session time to be considered.

An example is: BEGIN=1136073600, End=1167631200, the begin corresponds to 01/01/2006 and the end corresponds to 01/01/2007.

REQUEST=2	:	LENGTH=2	:	BEGIN=b	:	END=e
				int		int

Table 5.8: First request terminal information

**First answer** The answer is composed of three fields that are repeated for all the attackers open sessions within the interval of time.

- tty: is the name of the tty;
- begin: is the begin time of the session;
- end: is the end time of the session.

ANSWER	:	LENGTH	:	TYPE	:	tty	:	begin	:	end
=2		=num		=result						
						text		int		int

Table 5.9: Regular first answer terminal information

num is the number of field information after the parameter type. It should be module 3.

**Second request** The number ‘3’ is reserved to perform this second request. Moreover we have other three fields.

- t: is the tty name. We want to show all the information about this tty;
- b: is the begin session time to be considered;
- e: is the end session time to be considered.

REQUEST	:	LENGTH	:	TTY	:	BEGIN	:	END
=3		=3		=t		=b		=e
				text		int		int

Table 5.10: Second request terminal information

An example is: TTY=pts1, BEGIN=1136073600, End=1167631200, LIMIT=5, the begin corresponds to 01/01/2006 and the end corresponds to 01/01/2007.

ANSWER=3	:	LENGTH=num	:	TYPE=result	:	buffer
text						

Table 5.11: Regular second answer terminal information

**Second answer** The answer is composed of one field that is repeated for all the terminal information.

- **buffer:** is all the terminal information that corresponds to all the things the attacker has typed on the keyboard.

num is the number of field information after the parameter type.

## 5.4 Data testing

Through the socket a lot of data is exchanged between the client and the server. For a good security and data coherence these are tested every time they enter from the net, in both client and server.

The data are tested at two different levels: the general data and the specific data of one functionality.

**General data** These are the data of the headers of our protocol<sup>1</sup>. The headers are common to all the packets exchanged between the client and the server, like the protocol specificate. The test is done on the type of the data, thus not strictly on the data value. Into a packet exchanged on the net the headers are always of the same type for all the functionalities treated.

In the request packet, see table 5.3, the REQUEST and the LENGTH parameters are integers. If we receive a request and these parameters are not integers we have to generate an error. The same is for the answer packet, see table 5.4, the ANSWER and the LENGTH headers are integer and the TYPE is a string.

Another thing that we test at this level is the consistency between the field LENGTH and the number of parameters. The mean of this field is the number of parameters that follow the headers. This number is not dependent from the functionality and so we can test it here. If the number of parameters is different from the number LENGTH we generate an error.

The fields of the packet should match exactly the fields of the protocol. This means that the packet must have the same number of headers of the protocol. If all the fields are correct but the field LENGTH or the field REQUEST, is not present, we generate an error.

The content of the header TYPE should be “result”. The field TYPE is received only at client side, so if the content is not result this means that

---

<sup>1</sup> See the precedent paragraph General.

to the server side of the net same error has occurred. Generate an error to the receiver is equivalent to have catch the error that the transmitter has generated. In this case of error we have same information on the field “message error”, see table 5.5.

So what we do in the general data test is a control on the meaning of the data and this is functionality independent. Moreover we check if the string that represent the packet is well formatted. There is a character that allow to understand the limit between one field and another. This is the character “.”. A field cannot be empty. Another character is “=”. This special character separates the header field in two parts. The first part identifies the field and the second part is the value. Both these parts cannot be empty.

At the client side all these controls are in the method *checkGeneralParameters()* of the class *ClientSock*, see figure 3.8. This respects the fact that we manage an independent functionality control. At the server side we do the controls in the method *run()* of the class *ServerSocketThread*, see the figure 3.4. For both, the control is done as soon as read from the socket.

**Peculiar data** These are the data that depend on the functionality that utilizes it. In the precedent paragraph, we analyze only the formatting of the packet and the headers, we check if the content is of the type specified in the protocol. Here we check that the header’s content is the right one. At this level we do not care about the check of the type of the parameters or if the field exists. These controls are already performed on the *checkGeneralParameters()* that is temporally precedent to the check on the peculiar data.

If we are performing the dictionary attack functionality, the header REQUEST should be exactly ‘1’, because this is the number that identify this functionality. The same is for the header ANSWER, the content should be ‘1’.

Once tested the headers, we can test all the parameters that depend on the protocol definition. In the case of dictionary attack, for example, we have three parameters in the query: begin, end and limit. These parameters should exist and should be integer otherwise we have to generate an error. The number of these parameters is strictly binded to the functionality like the type of the parameters too. In the functionality terminal information, for example, in one of the interaction, there are the parameters tty, begin and end, see table 5.10. tty is a text, begin and end are integer. As you can notice the tests we have to do are different from the one of the dictionary attack.

Another important thing to test in general check, is that the number of parameters returned from a query should be module N, like already mentioned in the precedent section. In the case of dictionary attack the number of parameters returned is  $N = 3$ : attempts, begin and end, see table 5.7.

These information are repeated for each dictionary attack retrieved from the database,  $M$  for example, so the number of fields returned should be  $X = N \cdot M$  that is  $X \% N = 0$ . If the number of attacks of the dictionary attack returned is  $M = 5$  then the number of the parameters returned is  $X = 15$ . The check is that  $X \% N = 0$ , so  $15 \% 3 = 0$ . If we have  $X \% N = 1$ , for example, this means that we have an attack that have one of the three information needed. For characterizing the attack we need all the three information. Following the example, the check on the LENGTH that we have explained before, is that the header LENGTH = 15 (this is done in the *checkGeneralParameters()*).

At the client side all these controls are in the method *checkDependentParameters()* of the class xxxThread that manage the interaction with the net, see figure 3.8, xxx is the name of the functionality. In the case of dictionary attack the class xxxThread is DictionaryThread. This location of the method respects the fact that we manage an independent functionality control.

At the server side we place the controls in the method *perform()* of the class that inherit from the class Functionality and that implements the peculiar functionality. In the case of dictionary attack the class is DictionaryAttack, see figure 3.3.2.

For both the control is done as soon as read from the socket, so before the beginning of the processing of the data.

## Chapter 6

# SQL query processing

In this chapter we speak about all concerning the database. The application interacts with the database to retrieve the information stored during the months of attack observation. The database used to store the observed data is in MySQL technology so the query and the drivers used from the high-interaction honeypot data processing system are related to MySQL.

We first introduce the file structure that allows to retrieve the SQL script. Then we speak about the tables manipulation of the dictionary attack in order to obtain the result and we show a numerical example. The table manipulation and the example is reported only for one functionality for sake of conciseness. At the end of the chapter we speak about the driver utilized to connect the application to the database.

### 6.1 General

For the usability of our software, it's very important to design SQL query with a good performance. In fact we have to care about the size of the database because it is big enough. The queries we have to perform are not trivial, we have two possibilities in its design. The easiest thing could be designing easy queries that provide partial results and then managing these in the layer above, the application layer. This solution is not very efficient from the performance point of view. In deed, same preliminary performance tests have proved that the response time in this case is of the order of minutes. What we need is to obtain directly, from the SQL, the answer for the client. This way the computation is quicker and we gain better performance.

In the future the system will be extended and new functionalities will be added, so we have to design it in a modular way. For each functionality we write in a separate text file the query of the database. The file will have a name concerning the query it will perform. In the case of the Dictionary attack functionality the name of the file will be "dictionary\_attack.txt". The file will contain a succession of SQL instructions that will return the result to

be sent to the client, so there will not be other data manipulations to do. The path of the file will be retrieved from another file called “queries\_file.txt”. This file contains all the path of the query file and it is formatted as follow:

```
4 c:\command_frequency.txt
1 c:\dictionary_attack.txt
2 c:\terminal1.txt
3 c:\terminal2.txt
5 c:\key_frequency.txt
```

In the example there are functionalities not implemented; this is just an example in order to understand the possible dispositions of the “queries\_file.txt”. It’s important to write the identification number because each functionality is associated to a number and the relative query is found utilizing this number. In the case of the terminal information functionality we have two files according to what we have said in the explanation of the server side organization, paragraph 3.3.2. In fact the two interactions will be treated independently and so the two sub-functionalities as two normal functionalities.

We don’t want an hard format restriction of this file. The only condition is that all the paths are disposed in different lines. Are allowed space lines and “TAB” for any suitable formatting of the file. So the idea is that for each client request we take the path of the query file from “queries\_file.txt” and then we execute the query. This way, if we want to add a new functionality,

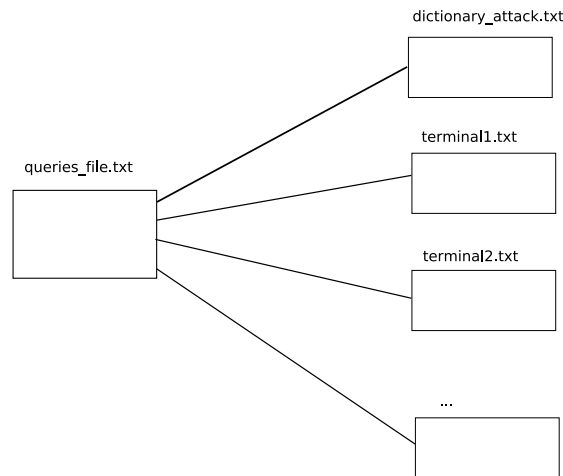


Figure 6.1: Image query schema

we have to create a new file that contains the query and add a new line in the queries file. There will not be a mix of SQL code that generally is error prone and the order of the new line in the “queries\_file.txt” is not important.

Once we have taken the query file, before the execution of the query, the file should be modified. In fact the query is parametric and it's not easy to manage with the conventional method of `PreparedStatement` class. This is because the query is composed of some other subquery and these will be executed separately. What we are calling query is a script containing several queries executed in sequence.

There will be many points of the script where we have to insert the parameters within the SQL code. To identify where we have to insert the parameters we use the notation `$PARAM$`. What we do is parsing the file before its execution and all parameters are changed with the correct values. From the client will arrive parameters in the form "PARAM=param", see chapter 5, and these will be matched with the parameters of the file. Also the generation of temporary tables is managed with parameters. The thread will create a pair "RANDOM=random" and this will take part of the set of the parameters as well. This way we allow a multi-thread context without caring of mistakes due to the same name in creation of tables. The tables in the file of the query should be called like this:

```
create table table$RANDOM$( . . . )
```

The user doesn't care about the real name of the temporary tables.

In the format of the file "queries\_file" we have no particular restrictions. It works like if we are running queries in MySQL batch mode. So it will be easier to write the query: when we write the file we can use comment, blank line and write the same query in more than one line. We can use also the "TAB" for formatting the file in a suitable manner, put space characters where we want and don't care about lower\upper case of the keys of the SQL query. In the following section we analyze the SQL query for each functionality we developed.

## 6.2 Dictionary attack

With this functionality we want obtain the IP address, see chapter 5, the number of attempts, the begin and the end of each attack.

The tables that are useful for retrieving this information are 'op\_hdr' and 'op\_sshd\_auth\_password' because these contain the field we need. The structures of these tables are given in table: 6.1 and table 6.2.

op_id	op_type	time	pid	index_op	op_size

Table 6.1: Table op\_hdr

op_id	is_auth	user_size	pass_size	ip_address_size	user	pass	ip_address

Table 6.2: Table op\_sshd\_auth\_password

We have to join these tables on the field ‘op\_id’ and in various steps to arrive to our result:

- We create a table called ‘table1’.  
This is created by an inner join between the two tables above selecting only the rows that have  $op\_hdr.op\_type = 6$  because we are interested only in the dictionary attack type and it is the number 6. We also have two more conditions; the time of the operations selected should be in a specific period of time that in the communication protocol is marked like begin end. We select only the IP and time field and create a primary key field. The result is the table 6.3.

id	time	ip

Table 6.3: Table table1

- We create a table called ‘table3’. This is created by an inner join between the precedent table1 with itself on the field  $table1.id = table2.id - 1$ . We also have two conditions more. We select only the rows with different IP and the rows with  $table2.time - table1.time > limit$ . This way we are closer to the information we want, we only keep significant rows that allow us to count the number of attempts with only a subtraction in the next step. In fact now we have in id a number that can

idt	id	time1	time2	ip1	ip2

Table 6.4: Table table3

be used to calculate the number of attempts. idt is the primary key generated automatically.

- Now we create a table called ‘table5’ that is the last table containing the final result. What we do is to do an inner join between the table3 and itself on the field  $table3.idt = (table4.idt - 1)$ . The field we take

are only ip, attempts and begin, end. We show clearly this join in the example the follows; we show all the fields of the join and then we will cut some column.

ip	attempts	begin	end

Table 6.5: Table table5

In this process we have lost two little parts of the information that we want. These two parts are at the begin and the end of the table3. This is shown in the example. We solve this problem adding some rows to the table5 and managing separately these cases.

### 6.3 Example: test

We create an example that allows to see if the total query is right. Instead of the first step, the join of table 'op\_hdr' and table 'op\_sshd\_auth\_password' we begin from a example table called 'table\_init' that contains ip and time of various attacks. The first join is easy to perform and doesn't need to be tested. The numbers utilized in the example are different from the reality but we have chose them to have a good test case, so looking to underly problems. We assume that the parameters of this example are:

- *begin* = 0;
- *end* = *infinite*;
- *limit* = 9.

It means that there is no selection for the time begin and end that is enough trivial to test in other way but we care about the time expired between two attacks. The entries are ordered by tps. The tps is the time instant of the attack, ip is the IP number of the attacker. We show it in a graphic way, figure 6.2. We put the begin of each new dictionary attack performed and all the attempts that form the attack. This paint is a particular instance of this functionality because depends on the parameters that the client has sent. All the attacks displayed are in the table init\_table.

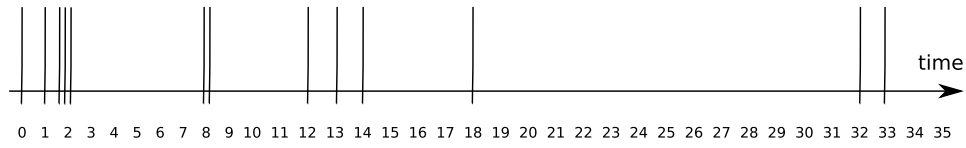
From this table we create the table1 adding the column of the identification, the primary key. The order is based on the pair (*time*, *ip*). We show it in a graphic way, 6.3. Here we paint it in different lines in order to have a clearer vision of the attacks split by IP. All the elements displayed are in the table1.

tps	ip
0	1
1	1
2	1
2	2
2	5
8	2
8	4
12	1
13	1
14	1
18	2
32	3
33	3

Table 6.6: Table init\_table

id	time	ip
1	0	1
2	1	1
3	2	1
4	2	2
5	2	5
6	8	2
7	8	4
8	12	1
9	13	1
10	14	1
11	18	2
12	32	3
13	33	3

Table 6.7: Table table1

Figure 6.2: Graphic representation of the table `init_table`

Now we do the self join of `table1` and we create the `table3`. Firstly we show the entire join and then we transform the table in function of the “where” clauses of the query described above. As we can see we have seven

attacks	idt	id	time1	time2	ip1	ip2
1	1	1	0	1	1	1
	2	2	1	2	1	1
2	3	3	2	12	1	1
	4	4	12	13	1	1
3	5	5	13	14	1	1
	6	6	14	2	1	2
4	7	7	2	8	2	2
	8	8	8	18	2	2
5	9	9	18	32	2	3
	10	10	32	33	3	3
6	11	11	33	8	3	4
	12	12	8	2	4	5

Table 6.8: Table `table3` without where clauses

attacks in this example. If we select the rows we obtain table 6.9. We show

idt	id	time1	time2	ip1	ip2
1	3	2	12	1	1
2	6	14	2	1	2
3	8	8	18	2	2
4	9	18	32	2	3
5	11	33	8	3	4
6	12	8	2	4	5

Table 6.9: Table `table3`

it in a graphic way, in figure 6.4. Not all the elements displayed are in the table `table5` but just the pairs that are highlighted with the blue color.

With this computation we have grouped the information in a suited way. The `id` is an information of interest because it is linked with the positions

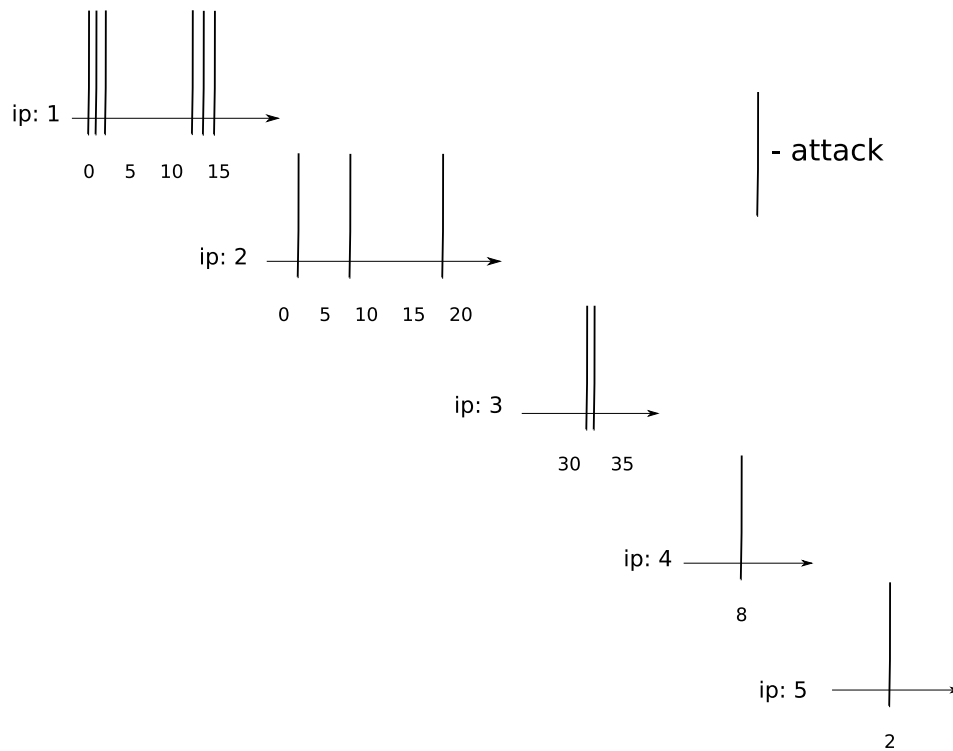


Figure 6.3: Graphic representation with different IP lines

of the attacks and so with the number of attempts. Moreover the times are important because they carry the information of the begin and the end of the attacks and we have also the IP. Now we do the self join of table3 and we create the table5. We can delete same columns that are not so important but firstly we show almost the whole join and we show it without any “where” clauses, table 6.10.

table3						table4				
idt	id	time1	time2	ip1	ip2	id	time1	time2	ip1	ip2
1	3	2	12	1	1	6	14	2	1	2
2	6	14	2	1	2	8	8	18	2	2
3	8	8	18	2	2	9	18	32	2	3
4	9	18	32	2	3	11	33	8	3	4
5	11	33	8	3	4	12	8	2	4	5

Table 6.10: Table table5 without where clauses

From this table we can take directly the information that we need:

- the number of attempts is given by the differences between table4.id

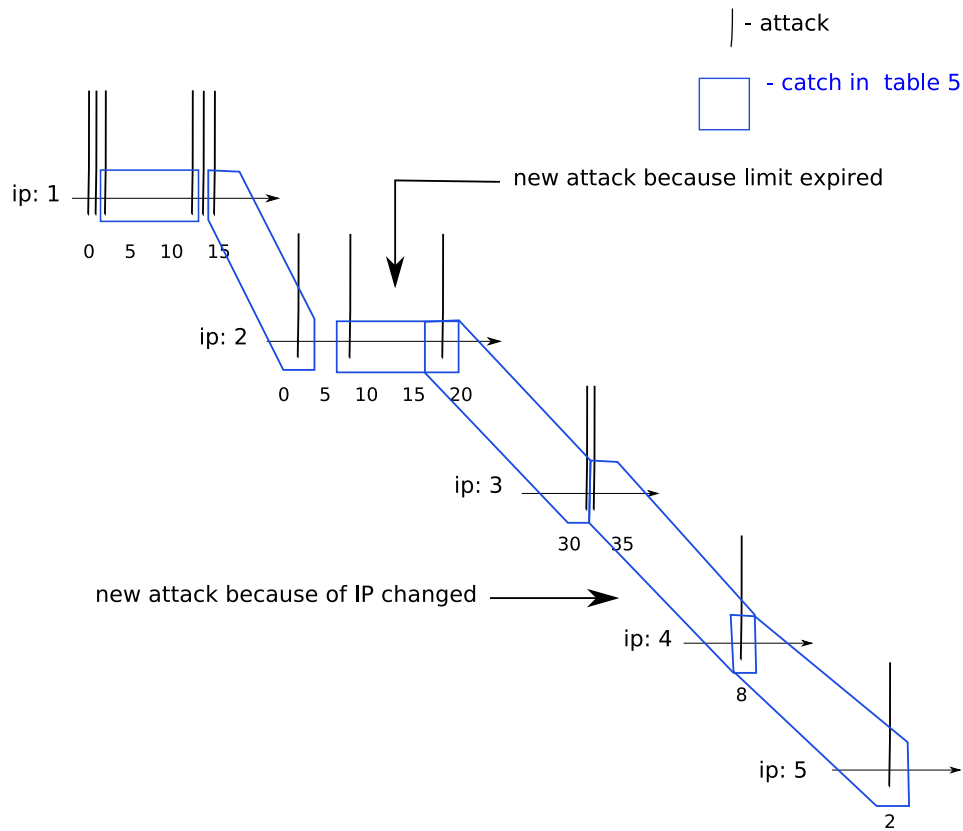


Figure 6.4: Graphic representation of table3

and table3.id;

- the attacker ip is table3.ip2;
- the begin is table3.time1;
- the end is table4.time1.

So what we have to do is to create a column called attempts and delete the column unnecessary inserting “where” clauses in the query. What we get is table 6.11 We show it in a graphic way in figure 6.5 All the elements highlighted with the color blue are in the table table5. The elements in green are missed from the computation of the query.

### 6.3.1 Special cases

From the table3 without “where” clauses we see that this way we have lost two attacks, the first and the last. This is shown in the figure above with

ip	attempts	begin	end
1	3	12	14
2	2	2	8
2	1	18	18
3	2	32	33
4	1	8	8

Table 6.11: Table table5

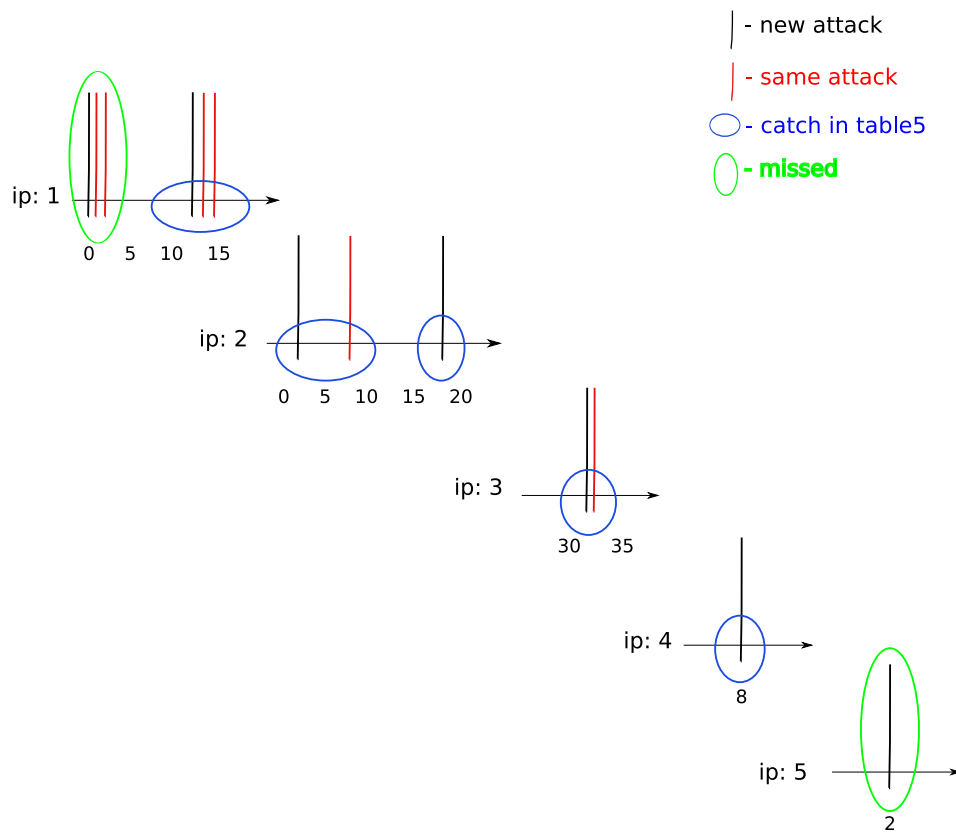


Figure 6.5: Graphic representation of all attacks caught and missed

circles. We treat these cases as particular cases.

**First attack** For this attack we do an insert taking:

- the ip of the table1 at the position  $id = 1$  that is the first position;
- the begin time from the table1.time at  $id = 1$ ;
- the attempts number, given by the id of table3.id;
- the end time from table3.time1.

So we will obtain the table 6.12 ordered by ip and begin.

ip	attempts	begin	end
1	3	0	2
1	3	12	14
2	2	2	8
2	1	18	18
3	2	32	33
4	1	8	8

Table 6.12: Table table5 with inserted first attack in red

**Last attack** For this attack, in table3 we have just the begin and the IP number in the position  $MAX(idt)$ . We use the table1 for retrieving the end information and the number of attempts in the position  $MAX(id)$ . So we do an insert taking:

- the ip of the table3;
- the begin time time2 from the table3;
- the attempts number, given by the subtraction  $table1.id - table3.id$ , this cover the general case;
- the end time from table1.

We will obtain the table 6.13 ordered by ip and begin. As we can see, from the last table and the table3 without the “where” clauses, all the attacks are detected in the right way.

ip	attempts	begin	end
1	3	0	2
1	3	12	14
2	2	2	8
2	1	18	18
3	2	32	33
4	1	8	8
5	1	2	2

Table 6.13: Table table5 with inserted lasts attacks in red

tps	ip
0	1
1	1
2	1

Table 6.14: Table init\_table anomaly one attack

**One attack present** However there is another case to consider that is not possible to see with this example. This case is called: “Anomaly of only one attack present”. If there is only one attack present with one or more attempts our code will not detect this attack. Consider the instance of table 6.14 for the table init\_table. With this instance table3 is empty, so also the first and the last special cases do not work because they are based on this table. What we do here is to create table 6.15. These are all the information that we need for the lost attack and we can take all from the table1 derived directly from table\_init:

- id1, ip1, time1: all the information of the first row of table1;
- id2, ip2, time2: all the information of the last row of the table1.

This is done with a creation of the tablex with the first row information and then with the update of the needed field with the last row. Then we insert this table, so the only row contained, inside the table5. There are two possibilities, the first is that we are in our instance case so we have added the write information, the second is that we are in the case of the first example made, so we have added a error information. We can see this second case checking same information in the table5. This case is valid only if the table5 has only one row, so only if the first row and the last row of the table have

id1	id	ip1	ip2	time1	time2
1	3	1	1	0	2

Table 6.15: Table tablex

the same ip, attempts and begin. What we do is delete the last row if this information in the first and the last row are different.

For the whole query see the annex.

## 6.4 Implementation environment: JDBC driver

In 1996, Sun released a version of the Java Database Connectivity (JDBC) kit. This package allowed programmers to use Java to connect, query, and update a database using the Structured Query Language (SQL). JDBC is an API for the Java programming language that defines how a client may access a database independently from the DBMS utilized. It provides methods for querying and updating data in a database. JDBC is oriented toward relational databases. The JDBC classes are contained in the Java package `java.sql` or in its extension `javax.sql`.

The use of Java with JDBC has advantages over other database programming environments. Programs developed with Java and JDBC are platform and vendor independent, i.e. the same Java program can run on a PC, a workstation, or a network computer. The database can be transferred from one vendor to another one and the same Java programs can be used without alteration.

**Driver JDBC or bridge JDBC-ODBC?** The idea behind JDBC is similar to Microsoft's Open Database Connectivity (ODBC). Both ODBC and JDBC are based on the X/Open standard for database connectivity. Programs written using the JDBC API communicate with a JDBC driver manager, which uses the current driver loaded. There are two architectures to communicate with the database utilizing a driver JDBC or a bridge JDBC-ODBC. In the first architecture, the JDBC driver communicates directly with the database. The driver connects to the database and executes SQL statements on behalf of the Java program. Results are sent back from the driver to the driver manager and finally to the application. In the second architecture, the JDBC driver communicates with an ODBC driver via a "bridge". A single JDBC driver can communicate with multiple ODBC drivers. Each of the ODBC drivers execute SQL statements for specific databases. The results are sent back up the chain as before.

The JDBC/ODBC bridge was developed to take advantage of the large number of ODBC enable data sources. The bridge converts JDBC calls to ODBC calls and passes them to the appropriate driver for the backend database. The advantage of this scheme is that applications can access data from multiple vendors. However, the performance of a JDBC/ODBC bridge is lower than a JDBC driver alone due to the added overhead. A database call must be translated from JDBC to ODBC to a native API. Moreover the driver JDBC is always optimized for the database we are using. Graphically

this is the situation for the two architectures:

1. Java → Driver JDBC → Database;
2. Java → Bridge JDBC-ODBC → Driver ODBC → Database.

It is for this reason that we utilize a JDBC driver in our application. In fact, as already said in the General section, the performance could be a problem for our query SQL, so it is fundamental for us obtain the better potentiality of the driver instead of the bridge.

The API provides a mechanism for dynamically loading the correct Java packages and registering them with the JDBC Driver Manager. The Driver Manager is used as a connection factory for creating JDBC connections.

In our application we utilize the database MySQL and the driver JDBC called Connector-J.

**Installation** The installation of the Connector-J for the Mysql database is simple. The copyright statement allows the redistribution of source and binary, also if is not exactly identical to the GNU license. We just have to add the directory in which the jar file was located to the CLASSPATH environment variable of the project. This completes the installation of the JDBC driver.

## Chapter 7

# Conclusion and future work

My stay of 6 months at LAAS has given me the possibility to work in a stimulating environment from a human and scientific point of view. I have learned several particular aspects of the research on security computing systems through the honeypots.

These systems work 24 hours for day from many months and they collect a huge quantity of data available for all the scientific community.

Future work will be oriented on changing the configuration of the high-interaction honeypots in order to observe other aspects not considered till now and store this information, the objective will be to add new vulnerability to the honeypot. Leaving more freedom to the attacker we can obtain more interesting information but it is very dangerous: the attacker could create damages not only on the honeypot but also outside of our controlled machine.

One interesting question could be: some ports are preferably chosen by script kiddies while others are reserved to some more elite attackers?

Another thing could be utilizing weaker password in order to see more people succeeding in breaking into the system.

Therefore, it would be interesting to keep doing this experiment over a longer period of time to see if things do change, for instance if a more efficient automation takes place.

The application that we have designed and implemented will serve to the scientific community in order to retrieve the data of the system who reside the central database. Through this application also people less expertize can access these information. The hope is that this new tool make the analysis of the data easier to the experts in term of legibility, practicality and clarity.

The terminal function could be utilized also for presenting the honeypot work. In fact, the database contains, for each command launched by the attacker terminal, the information of the time. This could lead to a simulation of the attacker operations on the terminal with the same velocity in typing and launching the commands.

In the future more functionalities will be implemented depending on the

needing of upcoming analysis. Could be interesting, i.e., modify the terminal functionality in order to obtain all the *wget* executed by the attacker in a particular session. This way we could know the files that the attacker has tried to download. The structure of the software is done in order to allow an easy upgrade of the software. In the structure of one functionality there are fixed stages to follow in order to create a new one.

Future work will be focused on the deployment of more high-interaction honeypot with a consequent exploitation of the collected data to better characterize attack scenarios and analyze their impact on the security of the target systems. The ultimate objective would be to build representative stochastic models that will enable us to evaluate the ability of computing systems to resist the attacks and to validate them based on real attack data.

# Appendix A

## Database structure

This annex presents the structure of the high-interaction honeypot database.  
This script creates all the tables.

```
use honeypot1;
CREATE TABLE op_hdr (
  op_id      INTEGER NOT NULL AUTO_INCREMENT,
  op_type    INTEGER,
  time       INTEGER,
  pid        INTEGER,
  index_op   INTEGER,
  op_size    INTEGER,
  PRIMARY KEY (op_id)
);

CREATE TABLE op_exec (
  op_id      INTEGER NOT NULL,
  filename_size INTEGER,
  tty_name_size INTEGER,
  arg_size   INTEGER,
  filename   TEXT,
  tty_name    TEXT,
  arg        TEXT,
  PRIMARY KEY (op_id)
);

CREATE TABLE op_sshd_auth_password (
  op_id      INTEGER NOT NULL,
  is_auth    INTEGER,
  user_size  INTEGER,
  password_size INTEGER,
  ip_address_size INTEGER,
```

```

        user            TEXT,
        password        TEXT,
        ip_address      TEXT,
PRIMARY KEY (op_id)
);

CREATE TABLE op_sshd_new_session (
    op_id              INTEGER NOT NULL,
    tty_name_size      INTEGER,
    ip_address_size    INTEGER,
    pw_name_size       INTEGER,
    tty_name           TEXT,
    ip_address         TEXT,
    pw_name            TEXT,
PRIMARY KEY (op_id)
);

CREATE TABLE op_tty_open_close (
    op_id              INTEGER NOT NULL,
    task_size          INTEGER,
    tty_name_size      INTEGER,
    task               TEXT,
    tty_name           TEXT,
PRIMARY KEY (op_id)
);

CREATE TABLE op_tty_read_write (
    op_id              INTEGER NOT NULL,
    buffer_size        INTEGER,
    task_size          INTEGER,
    tty_name_size      INTEGER,
    buffer             TEXT,
    task               TEXT,
    tty_name           TEXT,
PRIMARY KEY (op_id)
);

```

## Appendix B

### Dictionary attack query

This annex presents the SQL script executed in order to perform the first functionality: the dictionary attack. The result is obtained with different stages according to the section 6.2.

```
use honeypot;
```

```
create table if not exists table1$RANDOM$
(id integer not null auto_increment,
time integer, ip text, primary key(id))
select time, ip_address as ip from op_hdr
inner join op_sshd_auth_password
on op_hdr.op_id = op_sshd_auth_password.op_id
where (op_type = '6' and time < $END$ and
time > $BEGIN$)
order by ip, time;

create table if not exists table3$RANDOM$
(idt integer not null auto_increment, id integer,
time1 integer, time2 integer,
ip1 text, ip2 text, primary key(idt))
select t1.id as id, t1.time as time1, t2.time
as time2, t1.ip as ip1, t2.ip as ip2 from
table1$RANDOM$ as t1 inner join table1$RANDOM$
as t2 on t1.id = (t2.id - 1)
where ((t2.time - t1.time > $LIMIT$) or
(t1.ip <> t2.ip))
order by id;

create table if not exists table5$RANDOM$
(id integer not null auto_increment, ip text,
attempts integer, begin integer, end integer,
```

```

    primary key(id))
select t3.ip2 as ip, (t4.id - t3.id) as attempts,
       t3.time2 as begin, t4.time1 as end
from table3$RANDOM$ as t3 inner join
     table3$RANDOM$ as t4 on t3.idt = (t4.idt - 1)
order by t3.ip2;

```

Anomaly of the first attack:

```

insert into table5$RANDOM$ (ip, attempts, begin, end)
select table1$RANDOM$.ip as ip, table3$RANDOM$.id
       as attempts, table1$RANDOM$.time as begin,
       table3$RANDOM$.time1 as end
from table1$RANDOM$, table3$RANDOM$
where table1$RANDOM$.id = 1 and
      table3$RANDOM$.idt = 1;

```

Anomaly of last attack:

```

insert into table5$RANDOM$ (ip, attempts, begin, end)
select table3$RANDOM$.ip2 as ip,
       table1$RANDOM$.id - table3$RANDOM$.id,
       table3$RANDOM$.time2 as begin,
       table1$RANDOM$.time as end
from table3$RANDOM$, table1$RANDOM$
where (table3$RANDOM$.idt =
       (select max(table3$RANDOM$.idt) from
        table3$RANDOM$) and table1$RANDOM$.id =
       (select max(table1$RANDOM$.id) from
        table1$RANDOM$));

```

Anomaly of only one attack:

```

create table if not exists tablex$RANDOM$ (id1 integer,
      id2 integer, ip1 text, ip2 text, time1 integer,
      time2 integer)
select id as id1, time as time1, ip as ip1
from table1$RANDOM$ where table1$RANDOM$.id = 1;

update tablex$RANDOM$ set
id2= (select id from table1$RANDOM$ where
      table1$RANDOM$.id = (select max(table1$RANDOM$.id)
                          from table1$RANDOM$)), time2 =
(select time from table1$RANDOM$ where

```

```

        table1$RANDOM$.id = (select max(table1$RANDOM$.id)
        from table1$RANDOM$)), ip2 =
        (select ip from table1$RANDOM$ where
        table1$RANDOM$.id = (select max(table1$RANDOM$.id)
        from table1$RANDOM$)) where id1=1;

insert into table5$RANDOM$ (ip, attempts, begin, end)
select tablex$RANDOM$.ip1 as ip, tablex$RANDOM$.id2,
        tablex$RANDOM$.time1 as begin,
        tablex$RANDOM$.time2 as end
from tablex$RANDOM$ where
        (tablex$RANDOM$.ip1 = tablex$RANDOM$.ip2);

create table if not exists table6$RANDOM$
        (id integer not null auto_increment, ip text,
        attempts integer, begin integer,
        end integer, primary key(id))
select * from table5$RANDOM$;

delete from table6$RANDOM$ where
        id = (select max(id) from table5$RANDOM$) and ip =
        (select min(ip) from table5$RANDOM$) and begin =
        (select begin from table5$RANDOM$ where id = 1) and
        attempts <> (select attempts from table5$RANDOM$
        where id = 1);

```

Values returned:

```

select ip, attempts, begin, end from table6$RANDOM$
order by ip;

drop table table1$RANDOM$;
drop table table3$RANDOM$;
drop table table5$RANDOM$;
drop table table6$RANDOM$;
drop table tablex$RANDOM$;

```

## Appendix C

# Terminal information first interaction

This annex presents the SQL script executed in order to perform the second functionality: the terminal information. In the book only the stages of the dictionary attack functionality are explicated for sack of conciseness.

This functionality has a double interaction client-server. This code is relative to the first interaction.

```
use honeypot;

create table if not exists table1$RANDOM$ (time integer,
tty text, op integer)
select op_hdr.time as time,
op_sshd_new_session.tty_name as tty, 0 as op
from op_hdr inner join op_sshd_new_session
on op_hdr.op_id = op_sshd_new_session.op_id
where (time < $END$ and time > $BEGIN$);

create table if not exists table2$RANDOM$ (time integer,
tty text, op integer)
select op_hdr.time as time, op_tty_read_write.task
as tty, 1 as op
from op_hdr inner join op_tty_read_write on
op_hdr.op_id = op_tty_read_write.op_id
where (time < $END$ and time > $BEGIN$);
```

We have /dev/pts/# and we replace with pts#:

```
update table1$RANDOM$ set tty =
replace(tty, '2F6465762F7074732F', '707473');
```

```

create table if not exists table3$RANDOM$
  (id integer not null auto_increment, time integer,
  tty text, op integer, primary key(id))
  select * from table1$RANDOM$;

insert into table3$RANDOM$ (time, tty, op)
  select time, tty, op from table2$RANDOM$;

create table if not exists table4$RANDOM$
  (id integer not null auto_increment, time integer,
  tty text, op integer, primary key(id))
  select table3$RANDOM$.time, table3$RANDOM$.tty,
    table3$RANDOM$.op from table3$RANDOM$
  order by table3$RANDOM$.tty, table3$RANDOM$.time;

create table if not exists table5$RANDOM$
  (id integer not null auto_increment, id1 integer,
  time1 integer, tty1 text, op1 integer,
  id2 integer, time2 integer, tty2 text, op2 integer,
  primary key(id))
  select tx.id as id1, tx.time as time1, tx.tty as tty1,
    tx.op as op1, ty.id as id2, ty.time as time2,
    ty.tty as tty2, ty.op as op2
  from table4$RANDOM$ as tx inner join
  table4$RANDOM$ as ty on tx.id = (ty.id - 1)
  where (tx.op = 0) or (ty.op = 0);

create table if not exists table6$RANDOM$
  (tty text, time_begin integer, time_end integer)
  select tx.tty1 as tty, tx.time1 as time_begin,
    ty.time1 as time_end
  from table5$RANDOM$ as tx inner join table5$RANDOM$
  as ty on tx.id = (ty.id - 1)
  where (tx.op1 <> tx.op2) and (tx.op1 = 0)
  order by tx.tty1, tx.time1;

GENERAL: op1=op2=0 from table5$RANDOM$ (maybe more than one
row):

insert into table6$RANDOM$ (tty, time_begin, time_end)
  select tty1 as tty, time1 as time_begin,
    time1 as time_end
  from table5$RANDOM$ where op1 = op2;

```

LAST 1° case:  $op1 \neq op2$  from table5\$RANDOM\$ (only one row):

```
insert into table6$RANDOM$ (tty, time_begin, time_end)
select table5$RANDOM$.tty1 as tty,
      table5$RANDOM$.time1 as time_begin,
      table4$RANDOM$.time as time_end
from table5$RANDOM$, table4$RANDOM$
where (table5$RANDOM$.id1 + 1 <>
      ( select max(table4$RANDOM$.id)
        from table4$RANDOM$ )) and (table5$RANDOM$.op1
      <> table5$RANDOM$.op2) and (table4$RANDOM$.id
      = ( select max(table4$RANDOM$.id) from
        table4$RANDOM$ )) and (table5$RANDOM$.id1
      = ( select max(table5$RANDOM$.id1) from
        table5$RANDOM$ ));
```

LAST 2° case:  $op1 = op2$  from table5\$RANDOM\$ (only one row):

```
insert into table6$RANDOM$ (tty, time_begin, time_end)
select table5$RANDOM$.tty2 as tty,
      table5$RANDOM$.time2 as time_begin,
      table5$RANDOM$.time2 as time_end
from table5$RANDOM$ where (table5$RANDOM$.op1 =
      table5$RANDOM$.op2) and (table5$RANDOM$.id1 =
      ( select max(table5$RANDOM$.id1) from
        table5$RANDOM$ ));
```

```
select * from table6$RANDOM$;
```

```
drop table table1$RANDOM$;
drop table table2$RANDOM$;
drop table table3$RANDOM$;
drop table table4$RANDOM$;
drop table table5$RANDOM$;
drop table table6$RANDOM$;
```

## Appendix D

# Terminal information second interaction

This annex presents the SQL script executed in order to perform the second functionality: the terminal information. More precisely this code is relative to the second interaction with the server.

```
use honeypot;
```

```
select op_tty_read_write.buffer ,  
       op_tty_read_write.op_id  
from op_hdr inner join op_tty_read_write on  
       op_hdr.op_id = op_tty_read_write.op_id  
where (time < $END$ and time > $BEGIN$ and  
       task = '$TTY$');
```

## Appendix E

### Event channel

A new custom event must extend `EventObject`. Moreover, an event listener interface must be declared to allow objects to receive the new custom event. All listeners must extend from `EventListener`.

The event channel must register a set of listeners. The registration is done with the class `EventListenerList`. We see the example of one functionality, in the real application the number of methods `add()`, `remove()` and `fire()` depends from the number of functionalities. Also the number of `EventListenerList` depend from the number of the functionalities, we have a list for each functionality.

This example demonstrates all the steps necessary to create a new custom event, the associated listener and the `EventChannel`.

```
import javax.swing.event.EventListenerList ;

//Declare the event. It must extend EventObject.
public class MyEvent extends EventObject{
    public MyEvent(Object source){
        super(source);
    }
}

//Declare the listener class. It must extend
//EventListener. A class must implement
//this interface to get MyEvents.
public interface MyEvListener extends EventListener{
    public void myEventOccurred(MyEvent evt);
}

//Add the event registration and
//notification code to a class.
public class EventChannel{
```

```

        // Create the listener list
protected EventListenerList listenerList =
        new EventListenerList();

    //Allows classes to register for MyEvents
public void addMyEvListener(MyEvListener l){
        listenerList.add(MyEvListener.class, l);
    }
    //This methods allows classes to unregister
    //for MyEvents
public void removeMyEvListener(MyEvListener l){
        listenerList.remove(MyEvListener.class, l);
    }
    //This private class is used to fire MyEvents
void fireMyEvent(MyEvent evt){
        Object[] ls = listenerList.getListenerList();
        //Each listener occupies two elements:
        //the first is the listener class and
        //the second is the listener instance
        for (int i=0; i<ls.length; i+=2){
            if (ls[i]==MyEvListener.class){
                ((MyEvListener)ls[i+1]).myEventOccurred(evt);
            }
        }
    }
}

```

The lonely difference between the EventChannel of our application and thie EventChannel is the Singleton pattern. Here there is no Singleton pattern so in the real application when teh EventChannel is called the code is a bit different from the code that follow.

Here's an example of how to register for MyEvents.

```

EventChannel c = new EventChannel();

// Register for MyEvents from c
c.addMyEvListener(new MyEvListener(){
    public void myEventOccurred(MyEvent evt){
        // MyEvent was fired
    }
});

```

## Appendix F

### Example of dictionary

This file represent an example of dictionary used for SSH dictionary attack. It contains the attempts that an attacker does when he performs a brute force attack in order to establish a SSH connection. It is a very common dictionary file and it is often used in an automatic way from a script.

root asdfgh	admin admin	friends friends
root webadmin	admin admin123	friends friends123
root 12345678910	admin 123456	friends 123456
root r00t	admin administrator	master master
root com	administrator 123456	master master123
root id	tads tads123	master 123456
root 1234567	tads tads	amore amore
root asdfghjkl	tads tads123456	apples apples
root 0246	tip tip	apples apples123
root nevada	tip tip123	apples 123456
root router	tip 123456	apple apple
root 0249	myra myra	apple apple123
root l	myra myra123	apple 123456
root 1022	myra 123456	xxx xxx
root 10sne1	jack jack	xxx xxx123
root 111111	jack jack123	xxx 123456
root 121212	jack 123456	milller milller
root xxx	sya sya	milller milller123
root 1225	sya sya123	milller 123456
root 123	sya 123456	chicago chicago
root 123123	wang wang	Chicago chicago123
root 1234	wang wang123	Chicago 123456
root 12345	wang 123456	chipmast chipmast
root 1234qwer	marvin marvin	chipmast chipmast123
root 123abc	marvin marvin123	chipmast 123456
root 123go	marvin 123456	tweety tweety

root 1313	andres andres	tweety tweety123
root 131313	andres andres123	tweety 123456
root 13579	andres 123456	snoopy snoopy1
root 14430	barbara barbara	snoopy snoopy
root 1701d	barbara barbara123	snoopy snoopy123
root 1928	barbara 123456	snoopy 123456
root 1951	adine adine	ashley ashley1
root 1a2b3c	adine adine123	ashley ashley123
root 1p2o3i	adine 123456	ashley 123456
root 1q2w3e	test test	ashley ashley
root 1qw23e	test test123	bandit bandit1
root 1sanjose	test 123456	bandit bandit
root 2112	guest guest	bandit 123456
root 21122112	guest guest123	bandit bandit123
root 2222	guest 123456	madison madison
root 369	db db	madison madison123
root 4444	db db123	madison 123456
root 5252	db 123456	princess princess
root 54321	ahmed ahmed	princess princess123
root 5555	ahmed ahmed123	princess 123456
root 5683	ahmed 123456	viper viper
root 654321	alan alan	viper viper123
root 6969	albert albert	viper 123456
root 777	alberto alberto	billy billy
root 7777	alex alex	billy billy123
root 80486	alex alex123	billy 123456
root 8675309;	alex 123456	skkb skkb
root 92072	alfred 123456	faridah faridah
root 007	ali ali	faridah faridah123
root 123abc	ali 123456	fauzi fauzi123
root 007007	alice alice	fauzi fauzi
root 10sne1	alice alice123	fauzi 123456
root 4runner	alice 123456	fauzi 123456
root 2welcome	allan allan	ginger ginger
root *	allan allan123	ginger ginger123
root muiemare	allan 123456	ginger 123456
root tmp123	andi andi	cassie cassie
root qwerty	andi andi123	cassie cassie123
root administrator	andi 123456	cassie 123456
root root	andrew andrew	joefflores joefflores
root rootroot	andrew andrew123	joefflores joefflores123
root root1	andrew 123456	joefflores 123456
root 123456	amanda amanda	anthony anthony
root 1234567890	amanda amanda123	anthony anthony123

root qwerty	amanda 123456	anthony 123456
root administrator1	angie angie	jeffrey jeffrey
root admin	angie angie123	jeffrey jeffrey123
root backup	angie 123456	jeffrey 123456
root admin1	angela angela	superman superman
root secure	angela angela123	superman superman123
root secret	angela 123456	superman 123456
root passwd	anita anita	francis francis
root password	anita anita123	francis francis123
root password123	anita 123456	francis 123456
root 1a2b3c	anna anna	francois francois
root 1p2o3i	anna anna123	francois francois123
root 1q2w3e	anna 123456	francois 123456
root 1sanjose	arthur arthur	franklin franklin
root 2welcome	arthur arthur123	franklin franklin123
root welcome	arthur 123456	franklin 123456
root aaaaaa	aron aron	mortimer mortimer
root abcdef	aron aron123	mortimer mortimer123
root abcdefg	aron 123456	mortimer 123456
root action	austin austin	lloyd lloyd
root adidas	austin austin123	lloyd lloyd123
root airhead	austin 123456	lloyd 123456
root alaska	magic magic	guinness guinness
root amanda	magic magic123	guinness guinness123
root america	magic 123456	guinness 123456
root america1	bart bart	godzilla godzilla
carl carl	bart bart123	godzilla 123456
carl carl123	bart 123456	godzilla godzilla123
cesar cesar	ben ben123	charlott charlott123
corinna 123456	bind bind	wwwrun wwwrun
craig craig	bind bind123	www www
craig craig123	bind 123456	apache apache
craig 123456	bob bob	oracle oracle
cesar 123456	beny beny	apple1 apple1
clark clark	beny beny123	apple apple123
clark clark123	beny 123456	apple 123456
clark 123456	bert bert	netadmin netadmin
clinton clinton	bert bert123	netadmin netadmin123
clinton clinton123	bert 123456	netadmin 123456
clinton 123456	bill bill	scan scan
corinna corinna	bill bill123	scan scan123
corinna corinna123	bill 123456	scan 123456

# Bibliography

- [1] M. Kaâniche E. Alata, V. Nicomette and M. Dacier. Lessons learned from the deployment of a high-interaction honeypot. 2006.
- [2] Michael Bailey, Evan Cooke, Farnam Jahanian, and Jose Nazario. The internet motion sensor - a distributed blackhole monitoring system. In *Proceedings of NDSS 2005, Network and Distributed Systems Security Symposium, San Diego, USA*, 2005.
- [3] CAIDA Project. Home Page of the CAIDA Project, <http://www.caida.org>.
- [4] <http://www.dshield.org>. Home page of the DShield.org Distributed Intrusion Detection System, <http://www.honeynet.org>.
- [5] E Alata, M Dacier, Y Deswarte, M Kaaniche, K Kortchinsky, V Nicomette, Van Hau Pham, and Fabien Pouget. Collection and analysis of attack data based on honeypots deployed on the Internet. In *Proceedings of QOP 2005, 1st Workshop on Quality of Protection (collocated with ESORICS and METRICS), September 15, 2005, Milan, Italy*, Sep 2005.
- [6] Fabien Pouget, Marc Dacier, and Van Hau Pham. Leurre.com: on the advantages of deploying a large scale distributed honeypot platform. In *Proceedings of ECCE'05, E-Crime and Computer Conference, 29-30th March 2005, Monaco*, Mar 2005.
- [7] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, December 2005.
- [8] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of NDSS 2006, Network and Distributed Systems Security Symposium*, 2006.
- [9] Home page of Leurré.com: <http://www.leurre.org>.

- [10] Project Leurre.com. Publications web page: <http://www.leurrecom.fr/paper.htm>.
- [11] Dacier M., F. Pouget, and H. Debar. Honeypots: practical means to validate malicious fault assumptions. In *Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium*, pages 383–388, Tahiti, French Polynesia, 3-5 March 2004.
- [12] Fabien Pouget, Marc Dacier, and Van Hau Pham. Understanding threats: a prerequisite to enhance survivability of computing systems. In *Proceedings of IISW'04, International Infrastructure Survivability Workshop 2004, in conjunction with the 25th IEEE International Real-Time Systems Symposium (RTSS 04) December 5-8, 2004 Lisbonne, Portugal*, Dec 2004.
- [13] Michael Davis Edward Balas, Job de Haas. Available on: <http://www.honeynet.org/tools/sebek>.
- [14] Babès Jean Alberdi Ion and Le Jamtel Emilien. Uberlogger : un observatoire niveau noyau pour la lutte informatique défensive. In *Proceedings of SSTIC '05, Symposium sur la Sécurité des Technologies de l'Information et des Communications, June 1-3, 2005, Rennes, France*, Jun 2005.
- [15] Honeynet Project. Know your enemy: Defining virtual honeynets. Available on: <http://www.honeynet.org>.
- [16] Honeynet Project. Know your enemy: Learning with user-mode linux. Available on: <http://www.honeynet.org>.
- [17] Honeynet Project. Know your enemy: Learning with vmware. <http://www.honeynet.org>.
- [18] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Systems, Man and Cybernetics (SMC) Information Assurance Workshop. Proceedings from the Sixth Annual IEEE*, pages 29–36, 2005.
- [19] K. Kortchinsky. Patch for vmware. Available on: <http://honeynet.rstack.org/tools/vmpatch.c>.
- [20] Inc. VMware. Available on: <http://www.vmware.com>.
- [21] The PaX Team. Available on: <http://pax.grsecurity.net>.
- [22] Ross N. Williams. An extremely fast ziv-lempel data compression algorithm. In *Data Compression Conference*, pages 362–371, 1991.

- [23] EnergyMech team. Energymech. Available on: <http://www.energymech.net>.
- [24] US-CERT. Linux kernel mremap(2) system call does not properly check return value from do\_munmap() function. Available on: <http://www.kb.cert.org/vuls/id/981222>.
- [25] US-CERT. Linux kernel do\_brk() function contains integer overflow. Available on: <http://www.kb.cert.org/vuls/id/981222>.
- [26] Joseph Corey. Advanced honey pot identification and exploitation. Phrack, N 63, Available on: <http://www.phrack.org/fakes/p63/p63-0x09.txt>.
- [27] H. Kuno V. Machiraju G. Alonso, F. Casati. Web services. concepts, architectures and applications., 2004.
- [28] Kendall Scott Martin Fowler. Uml distilled: A brief guide to the standard object modeling language.