Experimental Validation of Architectural Solutions

Susanna Donatelli, Eric Alata,
João Antunes, Mohamed Kaâniche,
Nuno Neves, Vincent Nicomette,
Paulo Veríssimo

DI-FCUL TR–08–4

January 2008

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa
Portugal

Project no.:	**IST-FP6-STREP - 027513**
Project full title:	**Critical Utility InfrastructurAL Resilience**
Project Acronym:	**CRUTIAL**
Start date of the project:	**01/01/2006 Duration: 36 months**

Deliverable no.: D26

Title of the deliverable: Experimental validation of architectural solutions

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Contractual Date of Delivery to the CEC:	31/12/2007
Actual Date of Delivery to the CEC:	18/01/2008
Organisation name of lead contractor for this deliverable: CNIT	
Author(s): Susanna Donatelli[6], Eric Alata[4], João Antunes[5], Mohamed Kaâniche[4], Nuno Neves[5], Vincent Nicomette[4], Paulo Veríssimo[5]	
Participant(s):[6]CNIT, [4]LAAS-CNRS, [5]FCUL.	
Work package contributing to the deliverable:	WP5
Nature:	R
Dissemination level:	PU
Version:	3.0
Total number of pages:	

Abstract

This is a interim report on the experimental validation of architectural solutions performed in WP5. The two main contributions are the description of an attack injection tool for testing the architectural solutions and the description of a monitor and data collector that collects and analyses information about the behavior of the software after it has been attacked.

Keyword list: Attack injection, malicious behavior, monitor and data collection, honeypots, statistical analysis

DOCUMENT HISTORY

Date	Version	Status	Comments
29/10/2007	000	Draft	First version of table of contents
15/12/2007	001	Draft	First draft with call for contribution
21/12/2007	001	Draft	Contribution received
16/01/2008	002	Draft	Pre-final version distributed
18/01/2008	003	Submit	Submitted version

Table of Contents

1 INTRODUCTION

Identifying applications security related vulnerabilities and collecting real data to learn about the tools and strategies used by attackers to compromise target systems connected to the Internet is a necessary step in order to be able to build critical infrastructures and systems that are resilient to malicious threats. This deliverable presents two complementary types of experimental environments used in the context of CRUTIAL in order to fulfill these objectives. The first one concerns the development of a methodology and a tool (AJECT) for injecting attacks in order to reveal residual vulnerabilities in the applications and software components under study. This tool will be used in particular to locate security vulnerabilities in network servers and software components of the CRUTIAL reference architecture and information switches. The second type of experimental environment investigated in CRUTIAL and discussed in this deliverable concerns the development and the deployment of honeypots aimed at collecting data characterizing real attacks on the Internet. Such data are mainly used in the context of CRUTIAL to support the modeling activities carried out in WP2 and WP5 regarding the characterization and assessment of malicious threats. Useful feedback could be also provided to support design related activities, through the identification of common types, behaviors and scenarios of attacks observed on the Internet.

The structure of this Deliverable is as follows. Section 2 describes the methodology and the AJECT tool aimed at the identification of software security-related vulnerabilities and some preliminary experimental results illustrating the capabilities of the tool. Section 3 describes honeypot-based experimental environments investigated in CRUTIAL to collect and analyze real attack observed on the Internet. Conclusions and future work are drawn in Section 4.

2 SOFTWARE VULNERABILITIES IDENTIFICATION BASED ON ATTACK INJECTION

Applications have suffered dramatic improvements in terms of the offered functionality over the years. These enhancements were achieved in many cases with bigger software projects, which cannot be carried out by a single person or a small team. As a consequence, size and complexity has increased, and software development frequently involves several teams that need to cooperate and coordinate efforts. Additionally, to speedup the programming tasks, most projects resort to third-party software components (e.g., a cryptographic library, a PHP module, a compression library), which in many cases are poorly documented and supported. It is also not uncommon to re-use legacy code which was developed by people no longer available. All these factors contribute to the presence of vulnerabilities.

A vulnerability per se does not cause a security hazard, and in fact it can remain dormant for many years. An intrusion is only materialized when the right attack is discovered and applied to exploit a particular vulnerability. After an intrusion, the system might or might not fail, depending on its capabilities in dealing with the errors introduced by the adversary. Sometimes the intrusion can be tolerated [Veríssimo et al. 2003], but in the majority of the current systems, it leads almost immediately to the violation of its security properties (e.g., confidentiality or availability). Therefore, it is important to devise methods and means to remove vulnerabilities or even prevent them from appearing in the first place.

Vulnerability removal can be performed both during the development and operational phases. In the last case, besides helping to identify programming flaws which can later be corrected, it also assists the discovery of configuration errors. Intrusion prevention, such as vulnerability removal, has been advocated because it reduces the power of the attacker [Veríssimo et al. 2003]. In fact, even if the ultimate goal of zero vulnerabilities is never attained, vulnerability removal effectively reduces the number of entry points into the system, making the life of the adversary increasingly harder (and ideally discouraging further attacks).

In this section we describe a tool called AJECT which has been developed within the project. This tool uses an attack injection methodology to locate security vulnerabilities in software components, e.g., network servers running in the CRUTIAL Information Switches or in other interconnected machines. Some preliminary results with well known email servers are described to validate the capabilities of the tool.

The structure of this Section is as follows. Section 2.1 starts with the identification of the architectural components to be experimentally validated, Section 2.2 describes the attack injection tool, while Section 2.3 describes some preliminary experimental results illustrating the capabilities of the tool.

2.1 Identification of the run-time components to be validated and contribution of this interim report

The nature of the software and the reliance we place in it makes us even more vulnerable to deviations from its correct behavior. Critical infrastructures, such as the Power Grid for instance, have an important role in the normal functioning of the economy and general community, and thus they pose an even higher risk to the sustenance of the modern society pillars, such as the national and international security, governance, public health and safety, economy, and public confidence.

In recent years these systems evolved in several aspects that greatly increased their exposure to cyber-attacks coming from the Internet. Firstly, the computers, networks and protocols in those control systems are no longer proprietary but standard PCs and networks (e.g., wired and wireless Ethernet), and the protocols are often encapsulated on top of TCP/IP. Secondly, these networks are usually connected to the Internet indirectly through the corporate network or to other networks using modems and data links. Thirdly, several infrastructures are being interconnected creating a complexity that is hard to manage [van Eeten et al. 2006].

Therefore these infrastructures have a level of vulnerability similar to other systems connected to the Internet, but the socio-economic impact of their failure can be tremendous. This scenario, reinforced by several recent incidents [Wilson 2006, Amir et al. 2006], is generating a great concern about the security of these infrastructures, especially at government level.

The proposed reference architecture for critical infrastructures, models the whole infrastructure as a WAN-of-LANS, collectively protected by some special devices called CRUTIAL Information Switches (CIS). CIS devices collectively ensure that incoming/outgoing traffic satisfies the security policy of an organization in the face of accidents and attacks. However, they are not simple firewalls but distributed protection devices based on a sophisticated access control model. Likewise, they seek perpetual and unattended correct operation, which needs to be properly evaluated and validated. It is important that components, directly or indirectly, related to the correct functioning of the critical infrastructure are identified and validated.

In the next year, we will focus our validation efforts on two kinds of components. First, we will analyse the CIS devices since they are positioned at the border of the protected networks, and therefore are primary targets of attacks. Second we will look into servers that provide fundamental services to the other components of the network, namely Domain Name System (DNS) servers. If these servers have vulnerabilities that can be exploited by malicious adversaries, attacks to their security can potentially compromise the correct behavior of the critical network infrastructures.

2.2 The attack INJECTION tool

We propose attack injection with monitoring capabilities as a method for detecting vulnerabilities. This methodology tries to detect software bugs as an attacker would, i.e., trial and error, by consecutively attacking its target. Attack injection does not depend on a database of known vulnerabilities, but it rather relies on a generic and exhaustive set of tests.

Through careful and automated monitoring, the results of the attacks can be later analyzed to pinpoint the detected vulnerabilities. This allows the discovery of known and unknown vulnerabilities in an automated fashion.

We present and evaluate a vulnerability assessment tool called AJECT (***A**ttack in**JEC**tion **T**ool*) that implements the attack injection methodology. AJECT can be used for vulnerability detection and removal by simulating the behavior of an adversary. It injects attacks against a live system while observing its execution to determine if the attacks have caused a failure. In the affirmative case, this indicates that the attack was successful, which reveals the existence of a vulnerability. After the identification of the flaws, traditional debugging techniques can be employed to examine the application code and running environment, to find out the origin of the vulnerabilities and allow their subsequent elimination.

AJECT performs black box testing, so it does not require access to the source code to perform the attacks. However, in order to be able to generate intelligent attacks, AJECT has to obtain a specification of the protocol implemented by the target server (e.g., IMAP protocol specification for IMAP mail servers or HTTP protocol specification in case of web servers).

2.2.1 Using Attacks to Find Vulnerabilities

Every system's design and implementation should comply with a set of functional and/or non-functional properties, i.e., a service specification that describes its correct operation. The system is said to be *correct* if its service specification is not violated. But how much can one trust in that correctness? A system should give some guarantees that it will not fail. This measure is given by the system's *dependability*, which is the ability of a computing system to deliver service that can be justifiably trusted (Powell & Stroud 2002). Dependability aims at preventing the failure of the system, hence, in order to construct more dependable and reliable systems, one must know how the system and its components can remain correct, i.e., *how* and *why* do systems fail.

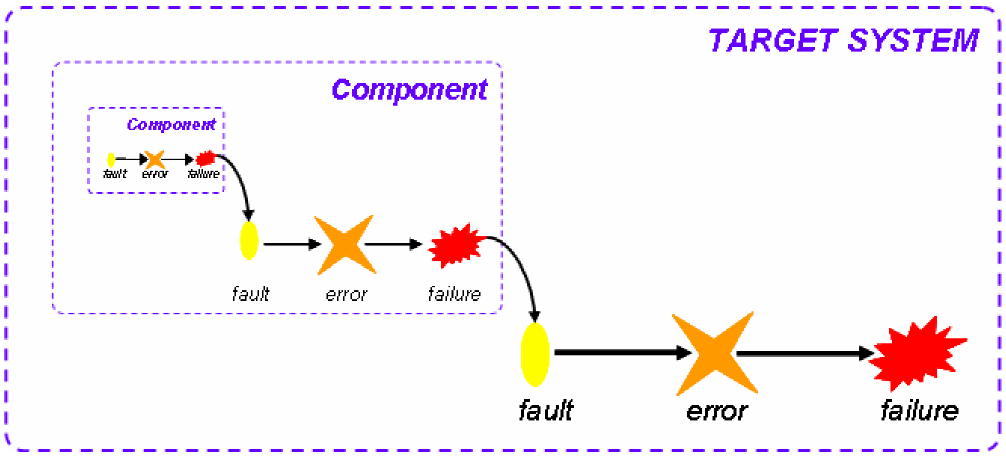


Figure 1: Fault, error, and failure.

The fault model presented in Figure 1 tries to explain how systems fail by following the sequence *fault* → *error* → *failure*. For instance, the purpose of a file system is to store and manage data records. These records, organized in computer files, are physically located in a storage device. The service specification of the file system defines, among other things, that any "read" operation will reflect the last "write" operation. Therefore, a record should always return the value of the last "write". A failure occurs when this specification is violated. To understand how these failures happen, one must study the process that explains how they appear. The cause, which is called the *fault*, can have an internal or external origin. For example, an electrical discharge (fault) can change the bits of some record, located in a specific disk sector. This fault can remain unnoticed, or dormant, until the record is read. The fault's manifestation is called an *error*, which in our example is the corrupted record. The *failure* of the system is the external observable effect of the error. If the file system lacks some sort of detection or correction mechanism for this type of error (e.g., checksums or redundancy) the record will return an incorrect value. This behavior clearly violates the service specification of the file system, or in another words, it represents the failure of the system.

However, the file system can be regarded as a component of a larger system, such as an operating system (OS). Therefore, from the OS point of view, the failure of the file system is seen as the fault of a component. So, as the figure shows, the fault–error–failure sequence is also a recursive model. If the affected disk record holds virtual memory data, such as a swap file, the file system failure (OS fault) will result in a memory page error. In turn, this error could freeze the entire system, leading to the failure of the OS.

However, the presence of the fault (or even the error) will not necessarily produce a failure. The system's correct operation can be maintained despite the presence of faults. If the system is supplied with fault detection or tolerance mechanisms, a greater dependability can be achieved.

Nevertheless, the type of faults that can arise are not reduced to accidental or arbitrary faults, like a physical defect or an electrical discharge. Faults can be much more complex and appear with higher probability. Intentional malicious faults are a good example. A potential intruder can leverage the failure probability by conducting a series of targeted attacks that can lead to the violation of the service specification. This type of faults does not follow any probabilistic distribution, nor any behavior pattern.

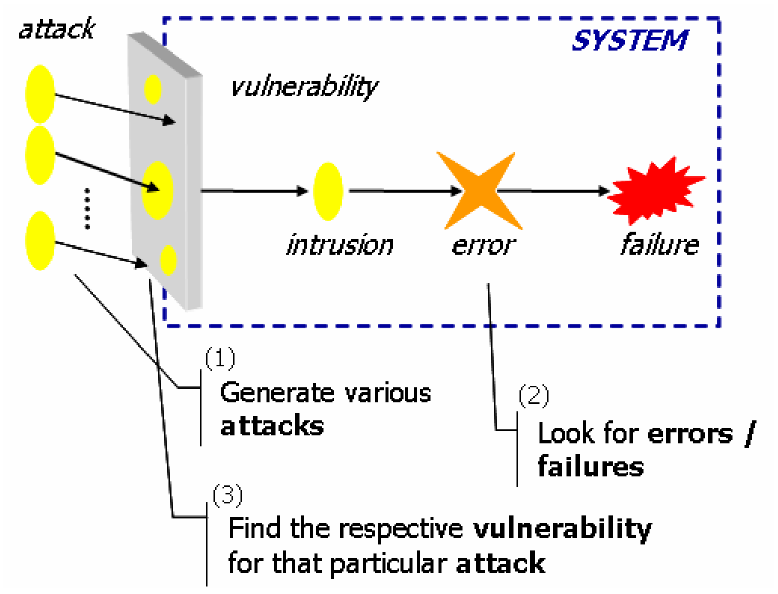

attack
SYSTEM
vulnerability
intrusion
error
failure
(1)
Generate various attacks
(2)
Look for errors / failures
(3)
Find the respective vulnerability for that particular attack

Figure 2: Composite fault model (attack, vulnerability, and intrusion).

The AVI (attack, vulnerability, intrusion) composite fault model, introduced in [Powell & Stroud 2002, Veríssimo et al. 2000], helps us understand the mechanisms of failure due to several classes of malicious faults (see Figure 2). It is a specialization of the fault–error–failure sequence applied to malicious faults – it limits the fault space of interest to the composition (*attack* + *vulnerability*) $\rightarrow$ *intrusion*. Let us analyze these fault classes. *Attacks* are malicious external activities, originating from outside the target system boundaries, that intentionally attempt to violate one or more security properties of the system – we can have an outsider or insider user of our network (e.g., an hacker or an administrator) trying to access sensitive information stored in a server. *Vulnerabilities* are usually introduced during the development phase of the system (e.g., a coding bug allowing a buffer overflow), or during operation (e.g., files with root setuid in UNIX). These faults can be inserted accidentally or deliberately, and with or without malicious intent. An attack that successfully activates a vulnerability causes an *intrusion*. This further step towards failure is normally succeeded by the production of an erroneous state in the system (e.g., a root shell or a new account with root privileges), and if nothing is done to process the error, a failure will follow.

The methodology utilized in the construction of AJECT emulates the behavior of an external adversary attempting to cause a failure in the target system. However, the goal of the attacks is not to exploit the system but rather to detect vulnerabilities that might permit that exploitation. The tool first generates a large number of attacks which it directs against the interface of the target (step 1, in Figure 2). A majority of these attacks are expected to be deflected by the validation mechanisms implemented in the interface, but a few of them might be able to succeed in exploiting a vulnerability and causing an intrusion. Some conditions contribute to increase the success probability of the attack. For example, a correct understanding of the interaction protocol used by the target eases the creation of more efficient attacks (e.g., it reduces the randomness of the tests); and a good knowledge about what type of vulnerabilities appear more frequently also helps to prioritize the attacks.

While the attacks are being carried out, AJECT monitors how the state of the system is evolving, looking for errors or failures (step 2). Whenever one of these problems is observed, it indicates that a new vulnerability has potentially been discovered. Depending on the collected evidence, it can indicate, with more or less certainty, that a vulnerability exists. For instance, there is a high confidence if the system crashes during (or after) the attack – this attack at least compromises the availability of the system. On the other hand, if what is observed is an abnormal creation of a large file, though it might not be a vulnerability – possibly related to a denial of service – it still needs to be further investigated.

After the discovery of a new vulnerability, there are several alternatives to deal with it, depending on the current stage of the development of the system (step 3). If the system is, for instance, in development, it is best to provide detailed information about the attack and the error/failure, so that a decision can be made about which corrective action should be taken (e.g., repair a software bug). On the other hand, if the tests are performed when the system is in live operation, then besides giving information about the problem, other actions might be worthwhile taking, such as automatically change the execution environment to remove the attack (e.g., by modifying some firewall rules) or shutdown the system until the administrator decides what to do.

In order to get a higher level of confidence about the absence of vulnerabilities in the system, i.e., its dependability, the attacks should be exhaustive and should exercise an extensive number of different classes of vulnerabilities. Still, one should also know that for complex systems it is infeasible to experiment all possible attack patterns, and therefore it is possible that some vulnerabilities remain undisclosed. Nevertheless, AJECT can be an important contributor for the construction of more secure systems because it mimics the malicious activities carried out by many hackers, allowing the discovery and subsequent removal of vulnerabilities before a real attempt is performed to compromise the system.

2.2.2 Architecture of the tool AJECT

The architecture of the AJECT tool is presented in Figure 3. The overall operation of the tool can be divided in the attack generation and the injection campaign.

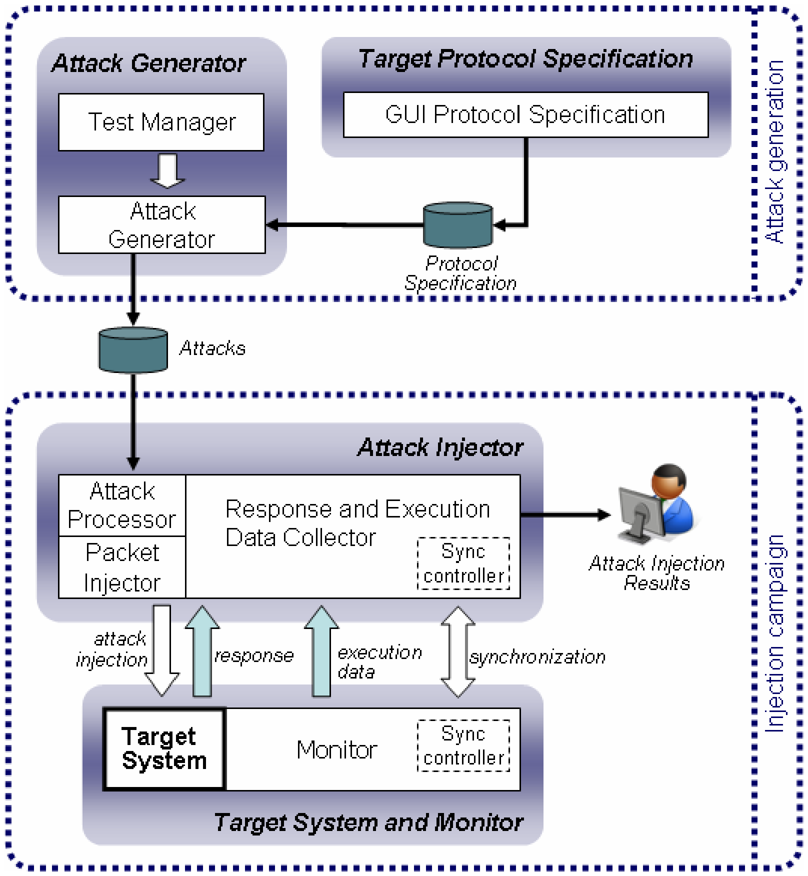


Figure 3: The architecture of the AJECT tool.

2.2.2.1 Test case generation phase

The attack generation is performed off-line and only once for each target communication protocol. The attacks can be used in the injection campaigns with any target system that shares the same application protocol (e.g., DNS, FTP). The format of the protocol messages, their fields, and data types are defined through a graphical interface application (GUI Protocol Specification). Additionally, the protocol states and messages that allow the transition between states are also identified.

The attacks are then constructed, based on the protocol specification and on a test case generation algorithm (implemented in the Test Manager and Attack Generator). The algorithm creates variations of the protocol messages that test the target system's ability to cope with some erroneous attribute, such as a missing parameter or an illegal type of data. Each attack is composed by one or more messages, where a few of them might be state transition messages (i.e., the messages required to take the protocol to the state where the fault must be injected). All test cases, or attacks, are saved in a disk file that will be used by AJECT in the injection campaign.

Target Protocol Specification Component

The Target Protocol Specification component is used to create a formal specification of the communication protocol utilized by the Target System. This specification is essential for two reasons: First, AJECT needs to be capable of bringing the server application from the initial state to any other state of the protocol. The reason for this is because certain protocol messages are exclusive to a particular state, such as specific requests that are only valid *after* successfully transiting to an authenticated state. Second, the syntax of the messages must be known to AJECT since many non-trivial attacks can only be created if this information is available. The Target Protocol Specification component eases the tool of having a special module for each new target protocol. One protocol specification can be used to generate attacks to any server implementations that use that protocol.

Currently, the specification can be done with a graphical interface that allows the definition of a state and a flow graph of the protocol. For each state it is possible to identify which messages can be sent and their syntax. The output of this component is a file that formally describes the target protocol, which is later imported by the Attack Injector. The protocol specification file provides the Attack Generator component with the essential knowledge to create valid protocol messages and to correctly change to the different protocol states.

Attack Generator Component

The attacks to be injected in the Target System are generated by the Attack Generator component. A Test Manager module is responsible for the implementation of the test generation algorithm that will create message variations from the protocol specification. The attack injection methodology does not specify the type of test cases generated from the Test Manager. It is up to the Test Manager implementation to devise good test generation algorithms aimed at good source code and vulnerability coverage. The Attack Generator applies any of the Test Manager algorithms to the target protocol specification to create a large and exhaustive set of test cases, or attacks. Although the attacks are based on a particular protocol, they are not restricted to a specific Target System. The attacks are saved in disk file to be used by the Attack Injector to test any Target System that uses the same communication protocol, e.g., attacks generated from the IMAP specification can be used with any IMAP server.

2.2.2.2 Injection campaign

This phase runs the entire universe of the generated test cases by carrying out each attack injection with a fresh copy of the target server. The attack injection is performed only after the Target System is online and ready. Therefore after the connection is established with the Target System, the Attack Injector sends a predefined “ping” message[1] and waits for the reply. Once the Target System is prepared, the Attack Injector sends the attack packets to the Target System, while the Monitor Component closely observes the execution of the Target System. AJECT processes and outputs the attack injection results in a human readable format, which can be used by the developers to detect any abnormal behavior and to assist them afterwards, in the debugging process.

Attack Injector Component

The Attack Injector is composed of three modules: the Attack Processor, the Packet Injector, and the Response and Execution Data Collector.

After the Target System is restarted and ready, the Attack Processor decomposes each attack in its components, i.e., the state transition messages and the attack message itself. The Packet Injector sends the transition messages to the Target System, and once the

[1] This message is defined by the Target Protocol Specification and should correspond to an innocuous protocol message that the Target System is known to respond, e.g., a failed login or a request for the version information.

protocol is in the designated state, it injects the attack message into the network interface of the Target System. The tests are synchronized with the Monitor, so that the latter can restart the Target System's process (i.e., the server application) before each attack injection. This simple synchronization protocol assures that the Target System is properly monitored and guarantees the same test conditions throughout the experiments. The Target System execution data, acquired by the Monitor, and the contents of the responses are gathered by the Response and Execution Data Collector.

Monitor Component

The Monitor is an external application, residing alongside the Target System, equipped with several control and monitoring capabilities. The Monitor component is supposed not to be obtrusive or affect the Target System in any significant way, i.e., the Target System behavior should be identical in the presence or absence of the Monitor. However, it is expected that some additional overhead is introduced by this component.

The Monitor is in charge of setting up the entire testing environment in the Target System: it needs to start up the target application, perform all configuration actions, initiate the monitoring activities, and in the end, free any utilized resources (e.g., processes, memory, or disk space). A synchronization module allows the Monitor to coordinate both the injection and the monitoring to determine the beginning and ending of each experiment. The Monitor can therefore reset the Target System environment before each experiment to ensure that each attack is done under identical conditions and that there are no interferences among the attacks. This inherent independence of the experiments simplifies the identification of the attack that caused the problems, and consequently, the discovery of the vulnerability.

The Monitor closely observes the target's flow of control (e.g., by intercepting and logging any software exceptions) to detect if the Target System goes to any erroneous software state. Therefore, there are many interesting operational characteristics desirable for monitoring, such as *Segmentation Fault* exceptions (crash) or something more subtle like an unusual set of OS signals. In the same way, the supervision of the allocation of the system resources, during the target's execution, can also be helpful to detect abnormal behavior activity, which may be indicative of the presence of a resource exhaustion vulnerability. This task is highly dependent on the mechanisms available in the local operating system (e.g., the ability to catch signals, such as memory segmentation errors, in UNIX). It is expected that the type of vulnerabilities AJECT is able to diagnose, is related to the type and detail of the collected information.

The architecture of the tool was defined to achieve two main purposes, the automatic injection of attacks and the data collection for analysis. However, its design was done in such a way that there is a clear separation between the implementation of these two goals. On one hand, in order to obtain extensive information about the execution, a proximity relation between AJECT and the target is required. Therefore, the Monitor needs to run in the same machine as the server application, where it can use low level operating system functions to gather, for example, statistics about the CPU or memory usage. On the other hand, the injection of attacks can usually be performed from a different machine. In fact, this is a desirable situation since it is convenient to maintain the target as independent as possible from the Injector, so that interference is kept to the minimum.

2.2.3 Implementation details of AJECT

The modular design of the architecture of AJECT provides the tool with a strong independence at various levels. The design is not platform specific, so it can be implemented in any operating system (OS) or hardware architecture. Moreover, since the Attack

Generator, Attack Injector, and Monitor components are inter-independent, new test classes or new monitoring capabilities can be added without interfering with one another.

There is, however, one restriction with the Monitor component. The Target System operates in a operating system that is of the utmost importance to the Monitor. The Monitor's dependence on the Target System is such that, due to the OS support, both the Monitor and the target application are required to run on the same machine.

AJECT is mainly implemented in Java. However, the Monitor was written in C++, because of the low-level operations it provides. C++ features specific OS low-level functionality, essential for controlling and monitoring the target process. In spite of the fact that C++ is an object-oriented and a high-level programming language, it lacks an important feature that Java possesses – a Java program can run similarly, and unmodified, on any Java virtual machine. This allows AJECT, except the Monitor component, to run on virtually any platform. The present section will give a more thorough insight on the current implementation of AJECT.

2.2.3.1 Target Protocol Specification

All attack generation and injection is totally independent of the intrinsics of the target protocol. The Target Protocol Specification component is responsible for the understanding of the protocol utilized by the target application. Without it, AJECT would not be able to create the protocol messages that will constitute an attack. Moreover, it is necessary for changing the different protocol states, from which the attacks must be launched.

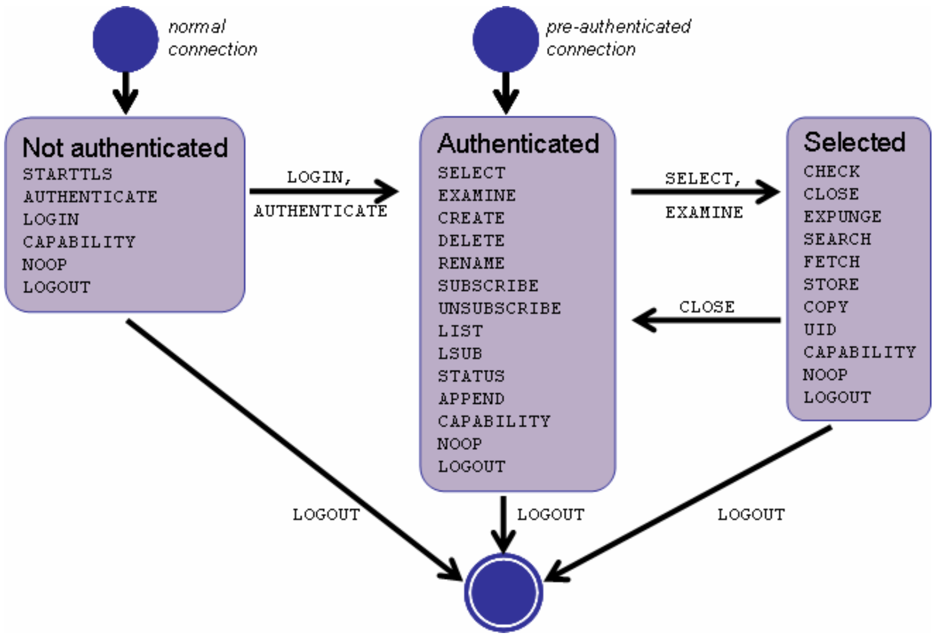
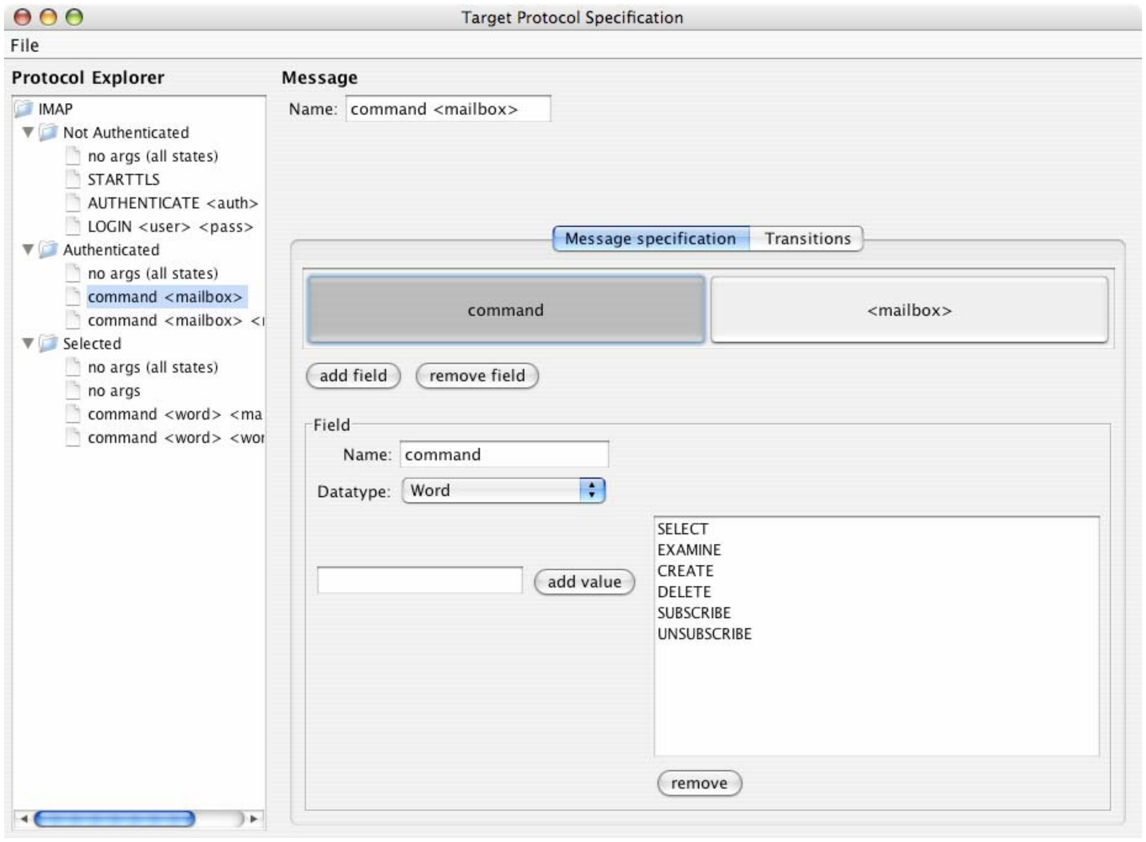


Figure 4: IMAP protocol finite state machine.



IMAP
- Not Authenticated
  - no args (all states)
  - STARTTLS
  - AUTHENTICATE <auth>
  - LOGIN <user> <pass>
- Authenticated
  - no args (all states)
  - command <mailbox>
  - command <mailbox> <
- Selected
  - no args (all states)
  - no args
  - command <word> <ma
  - command <word> <wor

Name: command <mailbox>

command
<mailbox>

Field
Name: command
Datatype: Word

SELECT
EXAMINE
CREATE
DELETE
SUBSCRIBE
UNSUBSCRIBE

Figure 5: Graphical user interface for the specification of the target protocol.

The target protocol can be regarded as a formal language, produced by a formal grammar or by a deterministic finite state machine. For example, the IMAP protocol can be described as a three-state deterministic finite automaton, as displayed in Figure 4. The boxes represent the IMAP protocol states, with its possible message types (i.e., IMAP commands): not authenticated, authenticated, and selected states. Some commands, if successfully executed, will trigger a state transition of the protocol (as depicted by the arrows).

A special graphical user interface application can be used to define the entire protocol specification, its states, message types, field data, and so on. A screenshot of the Target Protocol Specification is displayed in Figure 5. This particular snapshot depicts the definition of the IMAP messages, in the authenticated state, with two fields: a command string and a mailbox argument. The GUI application supports the definition of the legal type of data of each field, such as the valid strings in the field "command". However, the protocol specification is not restricted to textual protocols. Binary field data is also supported by AJECT. The GUI application provides several attributes of the binary fields, such as the size, if it is a signed or unsigned format, and the byte ordering (i.e., little endian or big endian).

2.2.3.2 Monitor

AJECT resorts to third-party libraries in the implementation of the Monitor component. The PTRACE facility is employed to intercept signals and any system call made by any of the server's processes (i.e., the main process, forked children, and threads). The target application is interrupted and its resource consumption probed, at the signal and system calls interception. This mode of operation passively traces the execution of the server, without much interference with its normal behavior.

Execution monitoring

A potential vulnerability is found if an abnormal behavior is detected on the Target System. The underlying OS offers some monitoring facilities that can be used to notice these irregularities. For instance, on UNIX machines there are OS functions for tracing the execution of a particular process, such as the PTRACE family functions used by some debuggers like GDB [GNU Foundation 2006]. These functions control the execution of a process by tracing the signals it receives. OS system calls can also be monitored the same way. The traced process is interrupted upon certain events, such as at the reception of a signal, or at the entry or exit point of a system call. Upon such an event, the Monitor (i.e., a dedicated execution thread[2]) intercepts the signal or system call and interrupts the traced process (i.e., target's application). The signals are logged for posterior analysis and the target's process is instructed to continue the execution. Unusual signals, such as a *Segmentation Fault*, are a very good indicator of the intolerance of a fault.

The current implementation of the Monitor is able to detect standard POSIX signals and system calls present in any UNIX-based machine, or derivatives, such as Linux or BSD. In order to use this monitoring method in other OSes, (e.g., Microsoft Windows), one needs to adapt it to the specific mechanisms these OSes signalize their exceptional software states (e.g., signals, exceptions, etc.).

Resource monitoring

Resource usage data is obtained at a few specific resource-related systems calls (e.g., memory utilization is probed after a memory allocation or de-allocation call). We have tried to reduce the overhead to a minimum by only updating the usage data at the relevant system calls. In some extreme situations, however, the monitoring activities can create some delays because of the constant pause, probe for data, and resume cycle.

The supervision of the system resources allocated during the target execution can be helpful to detect abnormal behavior which may be indicative of the presence of a vulnerability. For instance, if an application has suddenly allocated much more memory, it can be indicative of an erroneous state of memory starvation, or if the process is consuming a too great number of CPU cycles, it indicates a potential resource interlock.

The monitor maintains and regularly updates a global table with the resource usage data. The following local resources are watched:

- *total number of processes*, including forked children and threads of the target server. PTRACE signal interceptions are used to track new process ids (PIDs);
- *memory pages*, given by the number of resident set pages minus the shared pages, are obtained through the LibGTop library [Baulig & Kacar 2007];
- *file descriptors*, such as those identifying opened disk files or network sockets, are kept in a updated list of file descriptors. LSOF [Abell 2007] calls are used to keep track of the open files;
- *disk usage*, specified by the number of bytes written to disk. This value is obtained by parsing the LSOF's output for the files in use, and recording their size throughout the execution;

[2]A thread is also considered a lightweight process, or LWP.

- *CPU cycles*, corresponding to the work performed by the processor for all server's processes, is obtained with performance hardware counters. The linux kernel had to be patched to associate with each process a private set of *virtual* hardware counters [Pettersson 2002]. PREDATOR controls and accesses these counters through the PAPI library [London et al. 2001];
- *wall time*, measured as the elapsed time from the beginning of the main process execution, is computed by simple `gettimeofday()` calls. This resource is monitored mainly to compare its value with the number of CPU cycles. Large CPU and wall time discrepancies normally indicate a non-active wait, which suggests the presence of some timeout or deadlock.

3.4 Synchronization Protocol

To successfully diagnose vulnerabilities through attack injection, the tool must not only generate the actual attacks, but also observe its effects. Though it might seem a simple task, each attack must be carefully synchronized with its respective monitoring. Among the various tasks that result from this synchronization, there is:

- the execution of the target application prior to the attack (launched from the Monitor);
- the continuous supervision of the target (e.g., signal tracing, CPU usage, total number of allocated memory pages, etc.);
- the termination of the target application after the attack.

Synchronization protocol messages

The synchronization is accomplished through a simple protocol, composed by four types of messages, as represented in Figure 6:

- `SYNC_START` – message sent from the Injector, signalizing that a test is about to begin;
- `SYNC_END` – message sent from the Injector, signalizing that a test has ended;
- `SYNC_ACK` – message sent from the Monitor to acknowledge a received synchronization message and that the target application has been launched;
- `SYNC_DATA` – message sent from the Monitor, similar to the previous message, but appended with monitoring data.

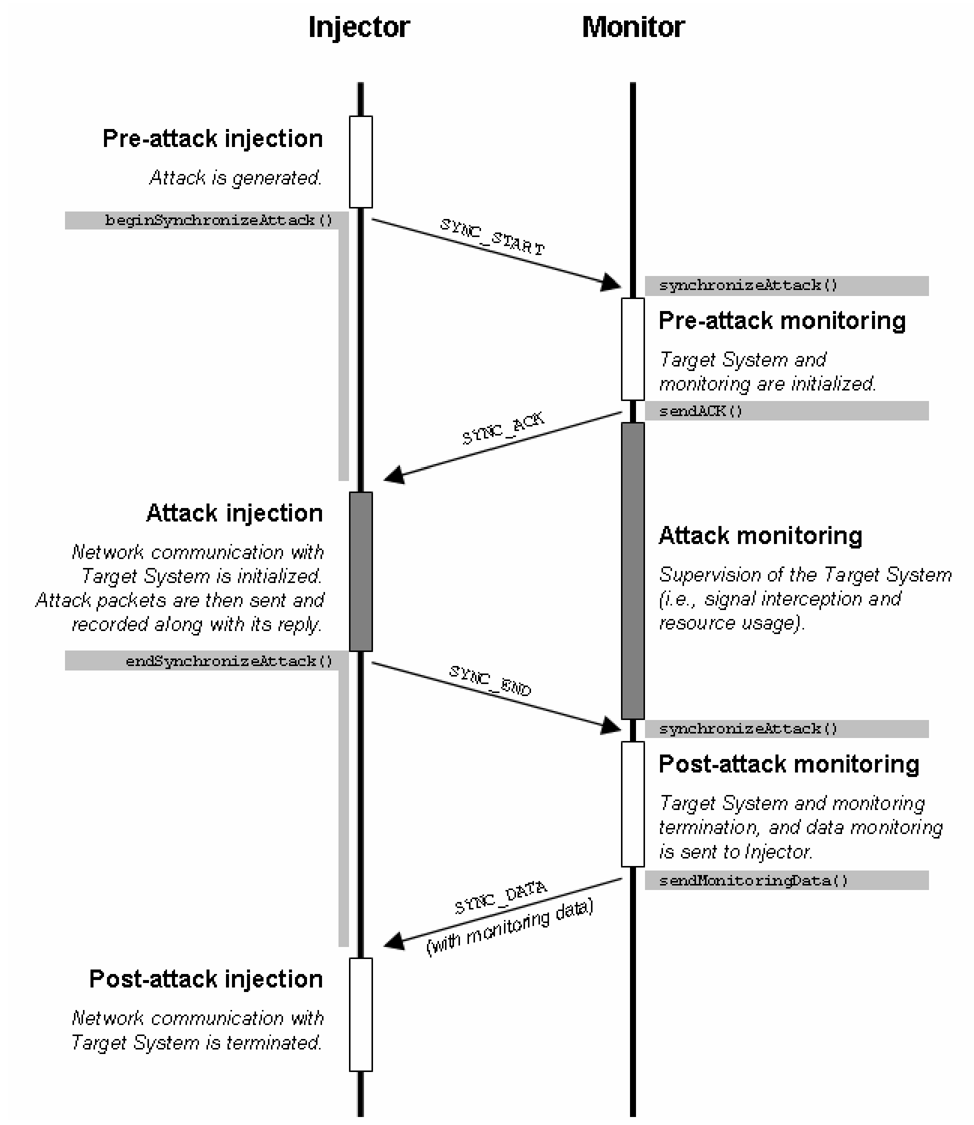

Injector
Monitor
Pre-attack injection
Attack is generated.
beginSynchronizeAttack()
SYNC_START
synchronizeAttack()
Pre-attack monitoring
Target System and monitoring are initialized.
sendACK()
SYNC_ACK
Attack injection
Network communication with Target System is initialized. Attack packets are then sent and recorded along with its reply.
Attack monitoring
Supervision of the Target System (i.e., signal interception and resource usage).
endSynchronizeAttack()
SYNC_END
synchronizeAttack()
Post-attack monitoring
Target System and monitoring termination, and data monitoring is sent to Injector.
sendMonitoringData()
SYNC_DATA (with monitoring data)
Post-attack injection
Network communication with Target System is terminated.

Figure 6: Synchronization protocol performed by the Attack Injector and the Monitor

Synchronization protocol between the Injector and the Monitor

The whole injection and monitoring of the attacks follows a three-step process attained by both the Injector and the Monitor (see Figure 6).

The *pre-attack injection/monitoring* starts when the Attack Injector processes an attack, from the attacks file, and sends a `SYNC_START` message to the Monitor. In turn, the Monitor replies with a `SYNC_ACK` after launching the target's process and starting the monitoring threads. The Attack Injector then knows that the Target System is prepared for the attack injection and is under the Monitor's supervision. The second step, the actual *attack injection/monitoring*, begins with the communication initialization between the Attack Injector and the Target System. The Attack Injector then proceeds by transmitting the attack messages. First, a set of transition messages to change the protocol state, and then, the actual attack packet. The target's process behavior (captured by the Monitor), and its responses to the attacks, are recorded for posterior analysis.

Finally, after receiving the attack's reply from the Target System (or after a timeout without any answer), AJECT reaches the *post-attack injection/monitoring* step. The Attack Injector sends a `SYNC_END` synchronization message to the Monitor and terminates all communication with the Target System. The Monitor will then *kill* the target's process, and terminate all monitoring and logging activity for that attack. The Monitor gathers all monitoring data for the attack and sends it to the Injector in a `SYNC_END` message.

2.2.3.3 Attack Tests

The tests, executed by the Injector, are performed by the injection of the attacks, generated by the test case generation algorithms in the Test Manager. Four different types of tests were implemented: a delimiter test, a syntax test, a value test, and a privileged access violation test. However, other different types of tests can be created, covering more classes of attacks, thus increasing the tool's capability to discover more vulnerabilities.

The current test generation algorithms are aimed at verifying if the Target System is able to cope with different kinds of protocol errors, namely:

- protocol messages with invalid, or missing delimiter characters;
- out-of-order, missing, or additional message fields;
- protocol messages with several kinds of invalid data (e.g., large or frontier values) or potentially dangerous data (e.g., information disclosure requests);

Each type of test, when applied to particular target protocol, generates a large number of attacks. The tool was also developed to support tests in a generic way, which means that more tests can easily be added to cover more kinds of attacks.

Delimiter test

Usually, applications are thoroughly tested for its normal and expected functionality, disregarding its robustness in dealing with malformed messages. This specific type of test plays with the delimiter fields of the protocol messages. These fields represent the delimiters of a particular field or packet. For example, the IMAP protocol messages end with a *carriage return* and *line feed* characters, while each field is delimited by *space* characters.

The current implementation of this type of test swaps and deletes the each of the delimiter fields. The generated attacks will consist of malformed protocol messages (i.e., with invalid or missing delimiters) but with valid data.

Syntax test

This kind of test generates attacks that infringe the syntax specification of the protocol as provided by the Target Protocol Specification. Example syntax violations consist on the addition, elimination, or re-ordering of certain fields of a protocol message specification.

This test regards a packet as a sequence of fields, each one occupying a certain number of bits. The type of data stored in a field is considered irrelevant, therefore, a 32-bit integer is deemed equivalent to any other type of data, such as 400 characters string. The main information required by the test is the size of every field, which is usually either predefined (e.g., it always occupies 4 bytes) or determined with some special control character (e.g., the space character serves as field terminator).

As an example, consider a message containing three different fields, which is represented as [A] [B] [C]. As an example, a few of the automatically generated attack packets that could be produced are:

- [A] [B];

- [A] [C];
- [A] [A] [B] [C];
- [A] [B] [A] [C];
- [A] [B] [C] [A];

As one can see, the fields remain unchanged, but it is their place in the message that changes, being removed or duplicated elsewhere.

Value test

The protocol specification also defines the type and validity of the data of the target protocol messages. This test class verifies if the target is able to cope with packets containing erroneous values.

An attack is generated in the following manner: each original protocol message is used to generate several attack packet variations; also, each field of this protocol message is iteratively chosen to be regarded as the *invalid field*; all the remaining fields will hold legal data values, while the invalid field is filled with malicious and illegal data. Since there are several fields in each packet, and each invalid field can take many non-legal values, this procedure can produce an overwhelming number of attacks. With the objective of keeping this number manageable, only a subset of the invalid data, hopefully representative of the whole set, is experimented.

As an example, consider a packet with two integer fields. The first field can only be set to 1, while the second field can take values between 0 and 1000. The first generated attacks would exercise different invalid values for the first field (e.g., -1, 0, and 1), while maintaining a legal value for the second value (e.g., 500). When all invalid data iterations are exhausted, the second field is chosen to be the invalid field. Then, several attacks are generated with the value 1 for the first field, and boundary and illegal values for the second (e.g., -1, 0, 1000, 1001, -100000, 100000). This test experiments different types of invalid values: almost valid (i.e., boundary values) and very invalid (i.e., large negative/positive integers).

However, there are several textual protocols, such as the IMAP protocol. Creating attacks for this type of protocols can be achieved by generating strings from different character combinations. The construction of these malicious strings is reasonably complex because it can easily lead to a combinatory explosion[3].

However, most of these character combinations are deemed equivalent with respect to the parsing and processing mechanisms of the target application, producing similar execution paths. So, a heuristic-based procedure was employed for the generation of the malicious data: first, a set of random tokens (fixed sized strings of random characters) is obtained; then, a set of specified malicious tokens (e.g., “%c”, “%x”, “””) and of joining tokens (e.g., “\”, space or none) is chosen. A large number of strings is obtained from the combination of one or more types of these different tokens. The result are strings with most of the characteristics that are usually found in hacker’s exploits, such as large strings or strings with strange characters (e.g., format string specifiers).

These strings are later used in the invalid fields, hence testing the target’s robustness in coping with this type of malicious input.

Privileged access violation test

[3]Just think that a string with 10 characters can have 26^{10} different combinations, even if we limit ourselves to the a..z characters.

This type of test tries to induce the server in granting access to some privileged operation, such as getting secret (or private) data from the Target System, or even modifying it. These privileged operations are always associated with some form of data, such as files or directories. Such actions may involve reading some well-known file, or writing to a particular directory. The success of such protocol requests indicates the incorrect action of the server, and thus the presence of a vulnerability.

The information disclosure test follows the same attack generation rationale of the previous class of test, but with a different configuration. In this test, the number of random tokens is set to a minimum, while the set of malicious tokens and of joining tokens is carefully chosen. Good malicious tokens are directory path names, well-known filenames, and existing usernames. These tokens are then automatically combined with the previously chosen joining tokens, such as ".", "..", or "/". This combinations generate a large number of path names to known files, which it uses during the attack generation. If a response provides valid data for one of the malicious requests, then the server is probably performing some illegal action, such as disclosing some confidential information.

For example, consider the file "/etc/passwd" that contains the usernames and encrypted passwords on a Linux machine. Some of the names that could be tried in the attacks are: ["./../etc/passwd"]; ["./../../etc/passwd"]; ["./../../../etc/passwd"].

2.3 Experimental Validation

In this section we present the validation of the attack injection methodology by experimenting AJECT with several IMAP servers. First, a description of the basic foundation for the experiments is provided, such as the communication protocol, the hardware and software specifications, the different tests, etc. Then, we analyze the experimental results achieved with AJECT.

2.3.1.1 Experimental Framework

This section gives a brief overview of the IMAP communication protocol that is utilized by the servers under test. It also describes the classes of attacks that were tried by the injector, and provides some information about the testbed.

IMAP Protocol

The Internet Message Access Protocol (IMAP) is a popular method for accessing electronic mail and news messages maintained on a remote server [Crispin 2003]. This protocol is specially designed for users that need to view email messages from different computers since all management tasks are executed remotely without the need to transfer the messages back and forth between these computers and the server. A client program can manipulate remote message folders (mailboxes) in a way that is functionally equivalent to local folders. The IMAP protocol provides a extensive number of operations, which include: creation, deletion and renaming of mailboxes; checking for new messages; permanently removing messages; server-based RFC-2822 and MIME messages format parsing and searching; and selective fetching of message attributes and text for efficiency.

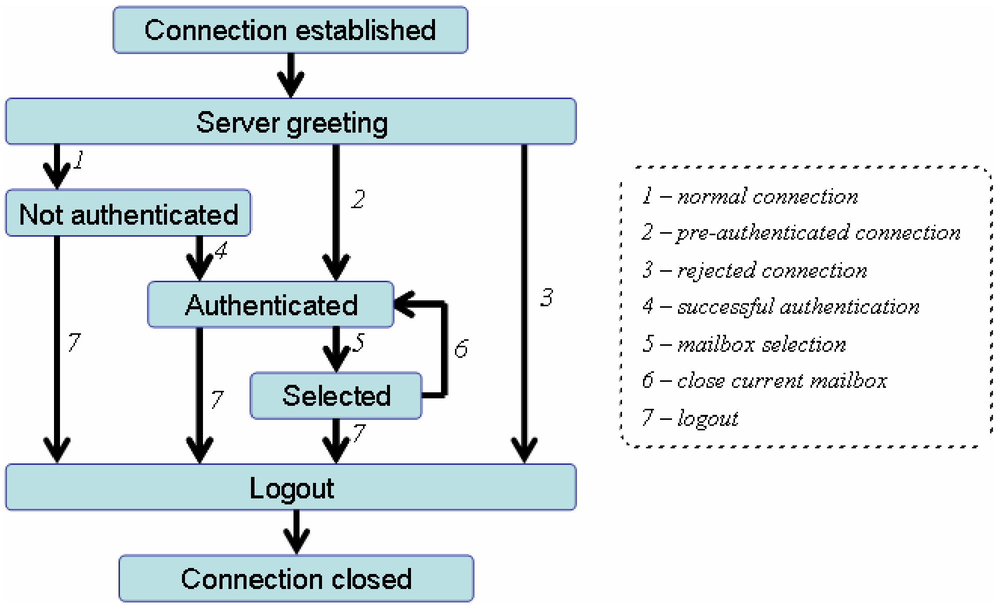


Figure 7: IMAP state and flow diagram

The client and server programs communicate through a reliable data stream (typically TCP) and the server listens for incoming connections on port 143. Once a connection is established, it goes into one of four states (see Figure 7). Normally, it starts in the *not authenticated* state, where most operations are forbidden. If the client is able to provide acceptable authentication credentials the connection goes to the *authenticated* state. Here, the client can choose a mailbox, hence transiting to the *selected* state, and execute the commands that will manipulate the messages. The connection goes to the *logout* state when the client indicates that it no longer wants to access the messages (by issuing a LOGOUT command) or when some exceptional unilateral action occurs (e.g., server shutdown).

All interactions exchanged between the client and server are in the form of strings that end with a CRLF (carriage return and line feed characters). The client initiates an operation by sending a command, which is prefixed with a distinct tag (e.g., a string A01, A02, etc). Depending on the type of command, the server response contains zero or more lines with data and status information, and ends with one of following completion results: OK (indicating success), NO (indicating failure), or BAD (indicating a protocol or syntax error). To simplify the matching between requests and responses, the completion result line is started with the same distinct tag provided in the client command.

Testbed and implementation issues

The experiments used several IMAP applications that were developed for different operating systems. Therefore, it was necessary to utilize a flexible testbed to ensure that the distinct requirements about the running environment could be accommodated. The testbed consisted of three PCs with Intel Pentium 4 at 2.80GHz and 512 MBytes of main memory. Two of the PCs corresponded to target systems, and each contained the IMAP applications and a Monitor. One of the machines could be booted in a few Linux flavors (e.g., Ubuntu, Fedora, and Suse) and the other on Windows (e.g., XP and 2000). The third PC ran the Injector components, collected the statistics, and performed the analysis of the results. This testbed configuration allowed for the parallel execution of two injection experiments (if needed, more PCs with target systems could easily be added, increasing the concurrency of the system).

At this moment, two Monitor components have been developed in C++, one for Linux and another for Windows. The Linux version implements all functionally that has been previously described, namely it collects a variety of execution data about the target (e.g., UNIX signals, resource usage) and synchronizes with the Injector. The Windows version is in an early stage of development, and it only provides basic functionality. Currently the Injector is capable of generating a large number of attacks for different test classes (e.g., syntax test, value test, privileged access violation test), and performs some level of analysis on the acquired execution data. The Java language was used in the implementation of the Injector to ensure that portability issues would not arise.

2.3.1.2 Experimental Results

The current section presents an evaluation of the vulnerability discovery capabilities of AJECT. This study executed several experiments to accomplish three main objectives: One goal was to confirm that AJECT is capable of catching a significant number of vulnerabilities automatically; A second goal was to demonstrate that different classes of vulnerabilities could be located with the tool, by taking advantage of the implemented tests; A third goal was to illustrate the generic nature of the tool, by showing that it can support attack injections on distinct IMAP server applications.

To achieve these objectives, we used AJECT to expose several vulnerabilities that were reported in the past in some IMAP products. Basically, the most well-known bug tracking sites were searched for IMAP vulnerabilities that were disclosed in 2005. The available vulnerable products were then obtained and installed in the testbed. The experiments consisted in using AJECT to attack these products, to determine if the tool could detect the flaws.

Another possible approach was to spend all our resources testing a small group of IMAP servers (one or two), trying to discover a new set of vulnerabilities. The experimental strategy presented in this thesis did not follow this approach because it would probably not allow to fulfill all experimental objectives. By lowering the number of different applications, and consequently of different development teams, the window of different classes of vulnerabilities would necessarily diminish. The same developers tend to make similar mistakes, so a larger spectrum of applications will probably contain different types of vulnerabilities.

Also, during the injection campaigns, AJECT was able to discover a new vulnerability, previously unknown to the security community.

Applications under test

To set up the experiments, vulnerability tracking sites – the *BugTraq* archive of `www.securityfocus.com`, and the *Common Vulnerabilities and Exposures* (CVE) database at `www.cve.mitre.org` – and several other hacker and security sites were searched for IMAP vulnerabilities. From this search it was possible to find 27 reports of security problems related to IMAP products during 2005. 7 of these reports were excluded because they proved themselves useless by not providing any specific information about the vulnerability itself.

Table 1: IMAP servers with vulnerabilities

ID	Application	OS	Date	Vuln. ID
A1	MailEnable Professional 1.54* and Enterprise Edition 1.04*	Win	Apr	CVE-2005-1014/5, CVE-2005-2278
A2	GNU Mailutils 0.6*	Lin	May	CVE-2005-1523
A3	E-POST Inc. SPA-PRO Mail @Solomon 4.0 4*	Win	Jun	BugTraq 13838/9
A4	Novell NetMail 3.52 B*	W/L	Jun	CVE-2005-1756/7/8
A5	TrueNorth eMailServer Corporate Edition 5.2.2*	Win	Jun	BugTraq 14065
A6	Alt-N MDaemon 8.0 3*	Win	Jul	BugTraq 14315/7
A7	GNU Mailutils 0.6.1*	Lin	Sep	CVE-2005-2878
A8	University of Washington Imap 2004f*	Lin	Oct	CVE-2005-2933
A9	Floosietek FTGate 4.4*	Win	Nov	BugTraq 15449
A10	Qualcomm Eudora WorldMail Server 3.0	Win	Nov	CVE-2005-3189
A11	MailEnable Professional 1.6 and Enterprise Edition 1.1	Win	Nov	BugTraq 15492/4
A12	MailEnable Professional 1.7 and Enterprise Edition 1.1	Win	Nov	BugTraq 15556

From the analysis of the remaining 20 reports, it was possible to identify 9 IMAP products with vulnerabilities. In a few cases, more than one version of the same application had problems. Table 1 provides a summary of these applications. For each product version, the table indicates our internal identifier (*ID*), the operating system where it runs (*OS*) and the date of the first report about a vulnerability (*Date*). Sometimes other reports appeared at a later time. Column *Vuln. ID* has the identifiers of the associated reports (i.e., CVE or BugTraq identifiers). For applications with multiple reports, it was used a condensed representation – for example, CVE-2005-1014/5 corresponds to CVE-2005-1014 and CVE-2005-1015.

There were two more products identified in the reports – the Ipswitch Collaboration Suite/IMail 8.13 and the Up-IMAPProxy 1.2.4. For the two products we were able to obtain the allegedly vulnerable versions and the exploits that were distributed by the hacker community. However, for some unknown reason, neither the AJECT tool nor the public available exploits were capable of exploring the described vulnerabilities. Therefore, we decided to disregard these products for further evaluation.

Vulnerability Assessment

After the identification of the flawed products, it was necessary to obtain as many applications (with the right versions) as possible. However, while attempting to obtain the reported (i.e., vulnerable) versions we met two main difficulties. First, in some cases these older versions were no longer available in the application's maintainers sites. This was especially true for commercial products, where whenever a new or patched version was produced, the older ones were removed. In most cases, where this older versions were not found in the official sites, a more thorough web search (e.g., using P2P networks) was found successful. A second problem was related to the cost of the commercial products. In these cases only the trial versions of the applications were available, which occasionally did not provide the required functionality for the discovery of the vulnerability.

Therefore, in order to assess AJECT, a different approach was employed for the unavailable applications. The Injector was used to generate and carry out the attacks against a dummy IMAP server. This simple server only stored the contents of the malicious packets received from the Injector, and returned simple responses. The packets were later analyzed to determine if one of the attacks could activate the reported vulnerability.

Table 2: Attacks generated by AJECT to detect IMAP vulnerabilities (**<A x *n*>** A repeated *n* times; **<OTHER-USER>** OTHER-USER is another existing username; ***** using CRAM-MD5 auth scheme)

ID	Vuln. Type	IMAP State	Potential Attack
A3	Access Violation	S2	A01 SELECT ./../../<OTHER-USER>/inbox
A4	Buffer Overflow	any	<A×2596>

a) Potentially detected vulnerabilities

ID	Vuln. Type	IMAP State	First Successful Attack
A1	Buffer Overflow	S2	A01 AUTHENTICATE <A×1296>
	Buffer Overflow	S2	A01 SELECT <A×1296>
A2	Format String	any	<%s×10>
A5	Format String	S2	A01 LIST <A×10> <%s×10>
A6	Buffer Overflow	S2	A01 CREATE <A×244>
	Buffer Overflow	any*	<A×1260>
A7	Format String	S3	A01 SEARCH TOPIC <%s×10>
A8	Buffer Overflow	S2	A01 SELECT "{localhost/user=\"}"
A9	Buffer Overflow	S2	A01 EXAMINE <A×300>
A10	Access Violation	S2	A01 SELECT ./../../<OTHER-USER>/inbox
A11	Buffer Overflow	S2	A01 SELECT <A×1296>
	Access Violation	S2	A01 CREATE /<A×10>
A12	Denial of Service	S2	A01 RENAME <A×10> <A×10>

b) Detected previous known vulnerabilities

Application	Vuln. Type	IMAP State	First Successful Attack
TrueNorth eMailServer Corporate Edition 5.3.4	Buffer Overflow	S3	A01 SEARCH <A×560>

c) New vulnerability discovered with AJECT

Table 2 presents some attacks generated by AJECT that successfully activated the software bugs present in the IMAP servers. Each line contains the internal application identifier (see Table 1), the type of bug, the IMAP state in which the attack was successful, and the attack itself. The attack injection campaigns were able to locate different types of bugs, including stack and heap buffer overflows, format strings, and information disclosure [Koziol et al. 2004], also see the next section). Information disclosure flaws may also allow other kind of attacks, especially if they could be explored with different IMAP commands, combined with write permissions. For example, a "CREATE pathname" command would allow the creation of a new file named "pathname".

The results of the experiments against the dummy IMAP server are shown in Table 2a. These two rows display the generated attacks that, supposedly, would activate the reported vulnerabilities.

The known vulnerabilities detected with AJECT are presented in Table 2b. Testing several different applications is very time consuming, because it involves performing an application survey. Besides the retrieving the applications, it also implicates the installation and configuration of the IMAP server. Moreover, each test could take a significant amount of time to complete. Therefore, we decided to carry out the injection campaigns only until the discovery of the first vulnerability of each application. The command corresponding to this first successful attack is presented in the last column of the table. In the few cases where experiments were left to run for a longer period, several distinct attacks were able to uncover the same problem. For example, after 24500 injections against the GNU Mailutils, there were already more than 200 attacks that similarly crashed the application.

Sometimes it was difficult to determine if distinct attacks were or not equivalent in terms of discovering the same flaw, especially in the cases where they used different IMAP commands. For example, if a bug is in the implementation of a validation routine that is called by the various commands, then the attacks would be equivalent. On the other hand, if no code was shared then there should be different bugs.

The equivalence of the attacks lies in the equivalence of the executed code instructions. If the attacks trigger the same vulnerability, i.e., the execution of the same piece of code, they are *equivalent*. However, different vulnerabilities are always detected by *non-equivalent attacks*, even the server's behavior is apparently similar. Actually, in order to find out the equivalence of the attacks, one would need to access the source code of the applications (something impossible to obtain for the majority of the products) and to monitor the instructions in real-time. Consequently, a more simplistic approach was taken: all successful attacks are deemed equivalent, except in the situations where the server's behavior or the attacks are obviously distinct, and therefore, correspond to different vulnerabilities.

During the course of our experiments, AJECT was also able to discover a previously unknown vulnerability as shown in Table 2c. The attack sends a large string in a SEARCH command that causes a crash in the server. This indicates that the bug is a boundary condition verification error, which corresponds to a buffer overflow. Several versions of the eMailServer application were tested, including the most recent one, and all of them were vulnerable to this attack.

Test Results

In Table 3 are represented the commands that were experimented in the various IMAP states. Some of the commands are very simple (e.g., composed by a single field) but others are much more intricate. As expected, the number of malicious packets generated from each command specification is proportional to its complexity.

Table 3: Commands tested in each IMAP state

Any State	**S1) Non Authenticated**
CAPABILITY	STARTTLS
NOOP	AUTHENTICATE <auth mechanism>
LOGOUT	LOGIN <username> <password>

S2) Authenticated	**S3) Selected**
SELECT <mbox>	CHECK
EXAMINE <mbox>	CLOSE
CREATE <mbox>	EXPUNGE
DELETE <mbox>	SEARCH [charset spec] <criteria...>
RENAME <mbox> <new name>	FETCH <seq set> <msg data | macro>
SUBSCRIBE <mbox>	STORE <seq set> <msg data> <value>
UNSUBSCRIBE <mbox>	COPY <seq set> <mbox>
LIST <reference> <mbox [wildcards]>	UID <COPY | FETCH |...> <args>
LSUB <reference> <mbox [wildcards]>	
STATUS <mbox> <status data items...>	
APPEND <mbox> [flag list] [date] <msg literal>	

The remainder of this section will provide some explanations for the attack injection results presented in Table 2.

Delimiter test

This class of test first retrieves the delimiters characters from the protocol specification. Each protocol field was separated by a space character, so this was the field's final delimiter. The initial tag (used in every protocol message) was defined as the packet's initial delimiter. So, in the Target Protocol Specification definition, "A001 " was specified as being the initial delimiter. For the message's final delimiter, the RFC-3501 [Crispin 2003] specifies the carriage return and line feed characters (or CRLF for short).

Attacks directed at the packet's initial delimiter resulted in the server assuming that the first field was the initial tag, when in fact was the IMAP command. Since the protocol command was being mistaken for the initial tag, these attacks were instantly rejected for *unknown command* reasons.

It was interesting to observe that most IMAP servers did not require the packet's final delimiter to be CRLF, but just CR or *newline*. When omitted, the servers concatenated the messages forming a larger message. This was not in conformance with the goal of the tool, which was to create single, independent, and easily reproducible attacks, instead of a strange conjunction of packets from different attacks.

The same concatenation behavior happened with the field's final delimiter. This time it would concatenate the fields, still maintaining the packet's integrity. However, no abnormal behavior was detected from any of the attacks generated from this class of test.

Syntax test

Another class of test that did not produce any detected abnormal behavior in the Target System was the syntax test. The generated attacks were very simple and were quickly dismissed by the parsing validation mechanisms.

Table 4: Syntax test attacks sample

Att. No.	Attack Packet	Description	
…			
328	SELECT	*removed field*	SELECT
329	/inbox	*removed field*	
330	/inbox SELECT /inbox	*duplicated field*	
331	SELECT SELECT /inbox	*duplicated field*	
332	SELECT /inbox /inbox	*duplicated field*	
333	SELECT /inbox SELECT	*duplicated field*	
334	SELECT SELECT	*rem. and dupl. field*	
335	/inbox /inbox	*rem. and dupl. field*	
336	EXAMINE	*removed field*	EXAMINE
337	/inbox	*removed field*	
338	/inbox EXAMINE /inbox	*duplicated field*	
…			

Table 4 shows a subset of the generated attacks using this test class. These example attacks are packet variations of the SELECT and EXAMINE commands. The field contents are kept unchanged, but they are removed or duplicated elsewhere.

By infringing the syntax of the protocol in such an obvious way, the attacks were immediately dismissed by the validation routines, so no vulnerabilities were detected by the syntax test.

Value test

This test was very successful in detecting buffer overflow, denial of service, and format string vulnerabilities, because it focused on the generation of malicious data (e.g., long strings or strings with format string specifiers).

The idea behind the attack generation is very simple. As explained earlier, a set of malicious and joining tokens was previously specified. Then, the value test will generate various combinations from these tokens with some random data. The resulting attacks are packets with some invalid fields that explore some characteristics usually found in hacker exploits.

With this class of test, AJECT was able to detect 11 known vulnerabilities: 7 buffer overflow, 1 denial of service, and 3 format string vulnerabilities. A new and previously unknown buffer overflow vulnerability was also detected with this test.

Privileged access violation test

The goal of this test is to generate protocol requests that induce the server into performing some privileged action, without the necessary credentials. Three IMAP servers were found vulnerable to these attacks that tried to get secret (or private) data from the target system, or even to modify it. Such information is usually found in the server's hard-disk or memory, and it can correspond, for instance, to passwords kept in a configuration file or in memory resident environment variables. Hence, this test resorts to some special tokens, such as well-known file, directory, and user names.

On the IMAP protocol there a few arguments of some commands that are used to name a file. For example, the *mbox* on the EXAMINE command refers to a mailbox, which is specified by its file system path. So, this is a very interesting field for information disclosure vulnerabilities, or more general access violations. Actually, both detected vulnerabilities were related to the mailbox field: an information disclosure vulnerability, and another that granted write access to any directory.

2.4 Summary of results

AJECT simulates the behavior of a malicious adversary by injecting different kinds of attacks against a target server. In parallel, it observes the running application in order to collect various information. This information is later analyzed to determine if the server executed incorrectly, which is a strong indication that a vulnerability exists. This methodology can be used to detect vulnerabilities in network services, present in critical infrastructures, such as the CIS devices, control and management systems, or any other network component essential to the correct and safe operation of the critical infrastructure.

The attack injection methodology, and well as its implementation, accomplished a modular design and architecture. In fact, AJECT is relatively portable to different systems, since the Injector runs on a Java virtual machine. However, the build of the Monitor component requires to run in the target system.

Experimental tests with IMAP servers were carried out to evaluate the usefulness of the tool. These experiments indicated that AJECT could be utilized to locate a significant number of distinct types of vulnerabilities (e.g., buffer overflows, format strings, and information disclosure bugs). In addition, AJECT was able to discover a new buffer overflow vulnerability.

Besides the good results achieved with the current implementation, new classes of test and protocol specifications can be created and accommodated into AJECT, increasing its vulnerability and target coverage.

Another important feature present in AJECT is its automatic operation. The tool performs automatic test-case generation (i.e., the creation of the attacks) and injection, while at the same time it launches, terminates, and monitors the target server.

3 HONEYPOT-BASED ARCHITECTURES

Besides performing controlled experiments to identify residual vulnerabilities in architectural and software components, we also need to collect data issued from real observations of attacks on the internet to improve our understanding of the behaviour of the attackers and their strategies to compromise the machines connected to the Internet. Also, such data is useful to elaborate statistical models and realistic assumptions about the occurrence of attacks, that are necessary for the evaluation of quantitative security measures. Some examples of such models are presented in deliverable D25 .

In this section, we present honeypot-based architectures that are aimed at fulfilling this objective. This section is structured as follows. Section 3.1 presents basic background and related work about honeypots. In particular, two types of honeypots offering different levels of interaction to the attackers are discussed. The architectures corresponding to each of these types of honeypots used in the context of CRUTIAL to support data collection are described in sections 3.2 and 3.3 respectively. Finally, section 3.4 describes the types of data that can be collected and possible exploitations of the data to characterize observed attack processes.

3.1 Background

Today, several solutions exist to monitor malicious traffic on the Internet, including viruses, worms, denial of service attacks, etc. An example of the proposed techniques consists in monitoring a very large number of unused IP address spaces by using the so called network telescopes [CAIDA, Moore *et al.* 2001], blackholes [Cook *et al.* 2004] or Internet Motion Sensors [Bailey *et al.* 2005]. Another approach used e.g., in the context of DShield [DShield] and the Internet Storm Center [ISC], consists in centralizing and analyzing firewall logs or intrusion detection systems alerts collected from different sources around the world. Other popular approaches that have received increasing interest in the last decade are based on honeypots. We can mention e.g., Leurré.com [Pouget *et al.* 2005], HoneyTank [Vanderavero *et al.* 2004], and many national initiatives set up in the context of the honeynet project alliance [Spitzner 2002].

A honeypot is a machine connected to the Internet that no one is supposed to use and whose value lies in being probed, attacked or compromised [Spitzner 2002]. 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. Thus of the activities recorded should correspond to malicious traffic. This is the main advantage of using honeypots compared to other techniques that consist in collecting and analysing the data logged by firewalls or routers as in DShield [DShield], where the information recorded is a mixture of normal and malicious traffic.

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 and cannot be used to compromise the honeypot or attack other machines on the Internet. 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 a 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, (some examples are presented e.g., [Artail *et al.* 2006, Provos & Holz 2007]).

Most of the currently deployed honeypots on the Internet are low interaction honeypots that are easy to implement and do not present any risk of being used by the attackers for attacking other machines. As an example, the Leurré.com data collection platform set up by Eurécom and to which LAAS contributes is based on the deployment of identically configured honeypots at various locations on the Internet (see Section 3.2).

In the context of CRUTIAL, we are interested in the analysis and the exploitation of data collected from both, low interaction and high interaction honeypots. The first type of honeypots is well suited to easily collect a large volume of data that can be used to characterize the time occurrence of the attacks and their distribution according to their type, origin, etc. In CRUTIAL, we rely on the data collected from the Leurré.com platform to carry out such analyses. The Leurré.com data collection platform is described in section 3.2. The second type of honeypots is needed to analyse the strategies and the behaviour of the attackers once they succeed in breaking into a target machine and try to progress in order to increase their privileges or carry out malicious actions. This requires the instrumentation of the honeypot with specific mechanisms dedicated to the capture and monitoring of attackers activities and to the control of their activities to prevent the use of the target machine as stepping stone for compromising other machines. In CRUTIAL, we have developed a specific high–interaction honeypot architecture dedicated to this purpose which is presented in Section 3.3.

3.2 Leurré.com data collection platform

The data collection environment *Leurré.com* is based on low-interaction honeypots using the freely available software called *honeyd* [Provos 2004]. Since 2003, 80 honeypot platforms have been progressively deployed on the Internet at various geographical locations. As illustrated in Figure 8, each platform emulates three computers running Linux RedHat, Windows 98 and Windows NT, respectively, and various services such as ftp, web, etc. All traffic received by or sent from each computer is saved in tcpdump files. A firewall ensures that connections cannot be initiated from the computers, only replies to external solicitations are allowed. All the honeypot platforms are centrally managed to ensure that they have exactly the same configuration. Every day, the data gathered by each platform are securely uploaded from a trusted machine during a short period of time 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. Integrity checks are also performed to ensure that the platform has not been compromised.

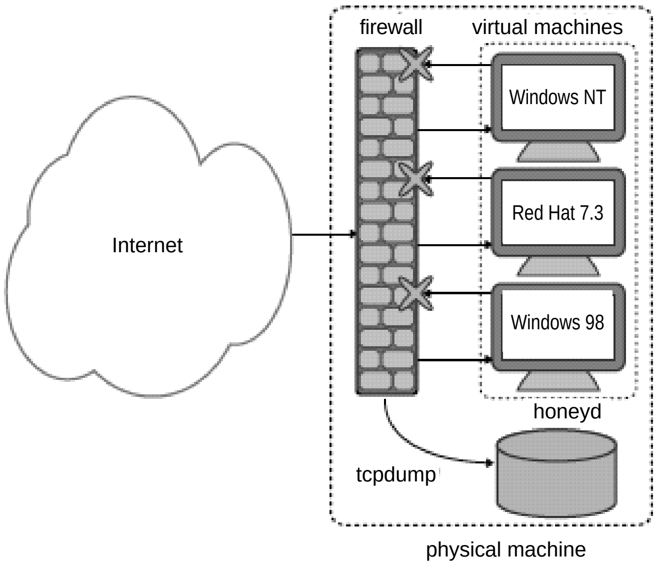


Figure 8- Leurré.com honeypot architecture

3.3 A high-interaction honeypot architecture

With low-interaction honeypots, the attackers can only scan ports and send requests to fake servers without ever succeeding in taking control over them. Thus, high-interaction honeypots are needed to allow us to learn about the behaviour of malicious attackers once they have managed to compromise and get access to a new host, and about their tools tactics and motives. We are mainly interested in observing the progress of real attack processes, and monitoring in a controlled environment the activities carried out by the attackers as they gain unauthorized access, capturing their keystrokes, recovering their tools, and learning about their motives.

The most obvious approach for building a high-interaction honeypot consists in using a physical machine and dedicating 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 capture and store the activities. Operating in the kernel is by far the most frequent manner to do it. Sebek [Provos & Holz 2007] and Uberlogger [Alberdi *et al.* 2005] operate in that way by using Linux Kernel Module (LKM) on Linux. More precisely, they launch a customised module to intercept interesting system calls in order to capture the activities of attackers. Data collected in the kernel is stored on a server through the network. Communications with the server are hidden on all installed honeypots.

Instead of deploying a physical machine that acts as a honeypot, a more cost effective and flexible approach would be to deploy a physical machine hosting several virtual machines that act as honeypots. Usually, VMware,User Mode Linux (UML) or, more recently, Qemu virtualisation and emulation software are used to set up such virtual honeypots. Some examples of virtual honeypots are presented in [Provos & Holz 2007].

In CRUTIAL, we have decided to build our own virtual high-interaction honeypot that can be easily customized to our needs and experiments. The design choices and the architecture of our honeypot are detailed in the following sections.

3.3.1 Objectives and design needs

More concretely, our objective is to set up and deploy an instrumented environment that will offer some possibilities to attackers to break into a target system under strict control, and will include mechanisms to log their activities. In particular, we are interested in capturing: 1) the communication traffic going through the honeypot over the network, 2) the keystrokes of the attackers on their terminals, 3) the logins and passwords use, and 4) the programs and tools executed on the honeypot.

The vulnerability to be exploited by the attacker to get access to the honeypot is not as crucial as the activity they carry out once they have broken into the host. That's why we chose for a first implementation to use a traditional vulnerability: weak passwords for `ssh` user accounts. Our honeypot should not be particularly hardened for two reasons. First, we are interested in analyzing the behavior of the attackers even when they exploit a buffer overflow and become root. So, if we use some kernel patch such as *Pax* [Pax 2007], our 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.

In our setup, only `ssh` connections to the virtual host should be authorized so that the attacker can exploit this vulnerability. On the other hand, any connection from the virtual host to the Internet should be blocked to avoid that intruders attack remote machines from the honeypot. This does not prevent the intruder from downloading code, using the `ssh` connection[4]. Forbidding outgoing connections is needed for liability reasons. This limitation precludes the possibility of observing complete attack scenarios. Moreover, it might also have an impact on attacker behavior: attackers might stop their attack and decide to never use again the honeypot for future malicious activities. To address this problem, some implementations limit the number of outgoing connections from the honeypot through the use of “rate limiting” mechanisms. Although this solution allows more information about the attack process to be captured, it does not address the liability concerns. A possible solution that we have investigated is to redirect outgoing connections to a local machine, while making the attackers believe that they are able to bounce from the honeypot. This solution is detailed in [Alata et al 2007].

As regards the mechanisms that need to be included in the honeypot to log attackers activities, capturing network traces using e.g., tcpdump as usually done in the case of low interaction honeypots would not be enough. We also need to record the activities carried out

[4] We have sometimes authorized http connections for a short time, by checking that the attackers were not trying to attack other remote hosts.

by the attackers on their terminal and the logins and passwords tried. This requires the modification of some OS calls as well as the `ssh` software. Additional mechanisms are also needed to regularly archive the information logged in a secured way.

In the next section, we describe the architecture of the proposed honeypot and describe the mechanisms that have been implemented for logging and archiving the activities of the attackers.

3.3.2 Architecture and implementation description

To fulfill the objectives listed in Section 3.3.1, we need an open source implementation of the target operating system. It is also important to be familiar with, and to have a deep knowledge of the selected operating system in order to be able to keep the activities of the attackers under strict control. For these reasons, we have decided to use GNU/Linux. As regards the implementation of the virtual machines, our choice was for a virtualisation software such as VMware or Qemu. Compared to VMware, Qemu presents the advantages of being freely distributed and open source. Indeed, we have developed a first implementation based on VMware that does not include the redirection mechanism. This implementation was then upgraded at a second stage to include the redirection mechanism, using Qemu. For both implementations, the honeypot was developed using a standard Gnu/Linux installation with kernel 2.6 with the usual binary tools. No additional software was installed except the `http apache` server. This kernel was modified as explained in the next subsection.

An overview of the general architecture of the honeypot is presented in Figure 9. The mechanisms implemented for capturing the attackers activities are highlighted. These mechanisms are described briefly in the following.

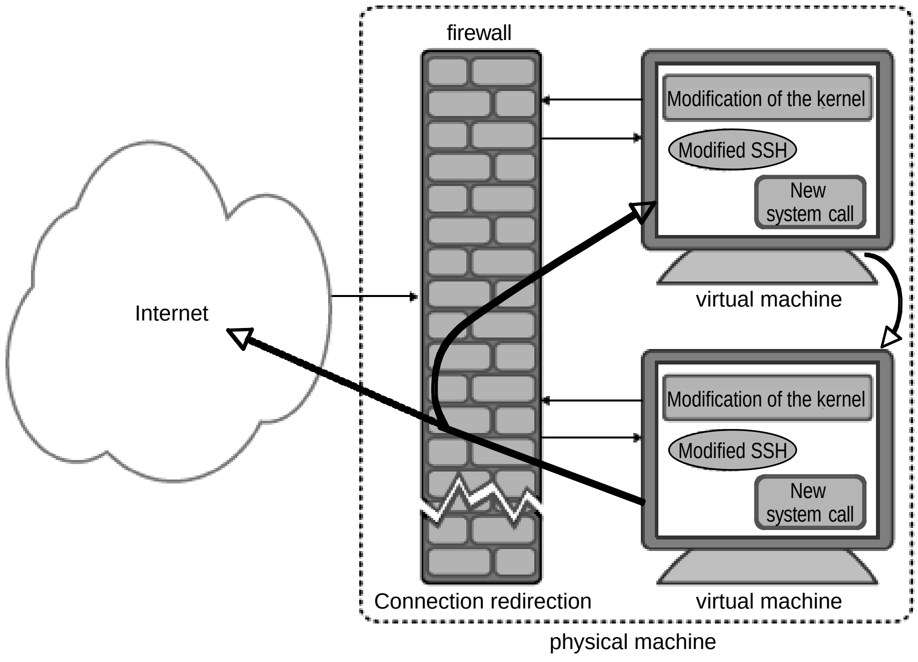


Figure 9- Overview of the high-interaction honeypot architecture

3.3.2.1 Data collection mechanisms

In order to log what the attackers do on the honeypot, we modified some drivers functions (`tty_read` and `tty_write`), as well as the `exec` system call in the Linux kernel. The modifications of `tty_read` and `tty_write` enable us to intercept the activity on all the terminals of the system. The modification of the `exec` system call enables us to record 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 of the honeypot itself. 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 we have studied, the information collected is directly sent to a remote host through the network. The advantage of our approach is that logging through the kernel is difficult to detect by the attacker (more difficult at least than detecting a network connection). It is noteworthy that the logging activity is executed on the real host not on the virtual, thus it is not easily detectable by the intruder (he cannot find anything suspicious in the list of processes for example). Furthermore, the data is compressed using the LZRW1 algorithm before being logged into the kernel memory.

Moreover, in order to record all the logins and passwords tried by the attackers to break into the honeypot we added a new system call into the kernel of the virtual operating system and we modified the source code of the `ssh` server so that it uses this new system call. The logins and passwords are 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, we do not have 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 analyses, mainly: i) the IP address of the attacking machine, ii) the login and the password tested, iii) the date of the connection, iv) the terminal associated (`tty`) to each connection, and v) each command used by the attacker.

3.3.2.2 Connection redirection mechanism

As indicated in Section 3.1.1, this mechanism is aimed at automatically and dynamically redirecting outgoing Internet connections from the honeypot to other local machines. The goal is to make the attacker believe he can connect from the honeypot to hosts on the Internet, whereas in reality, the connections are simply redirected towards another honeypot. The main idea is illustrated by the example presented in Figure 10.

In this example, *b*, *c* and *d* are honeypots and *a*, *e*, *f* and *g* are machines on the Internet. An attacker from Internet host *a* breaks into honeypot *b* (connection 1). From this honeypot, the attacker then tries to break into Internet host *e* thanks to connection 2. This connection is blocked by our mechanism. The attacker then tries another connection 3 towards Internet host *f*. This connection is accepted and automatically redirected towards honeypot *c*. The attacker is under the illusion that his connection to *f* has succeeded, whereas it has merely been redirected to another honeypot. The attacker tries to establish another connection 4 towards Internet host *g*. Similar to connection 3, this connection is accepted and automatically redirected towards honeypot *d*. The attacker finally initiates another connection (5) to Internet host *g* from host *f* (in reality, from host *c*). This connection is also accepted and is redirected towards honeypot *d*.

This mechanism allows the observation of attackers activity on different hosts. In general, a honeypot allows the activity of the attacker to be observed at only one side of the connection. The other connection end is the machine that interacts with the honeypot. For all redirected connections, we can observe an attacker on both connection ends.

On the other hand, it is possible for a clever attacker to see through the hoax. For example, in Figure 10, suppose the attacker already controls the machines a, e and f. He can then check, after establishing connection 3, if the machine he is connected to really is machine *f*. This limitation does exist; however we believe that many attackers will not systematically do such checks, in particular if the attack is carried out by non-sophisticated automatic scripts. Just as low-interaction honeypots provide some useful albeit limited information, more attack information would be gleaned from systems that implement our redirection mechanism than those that do not.

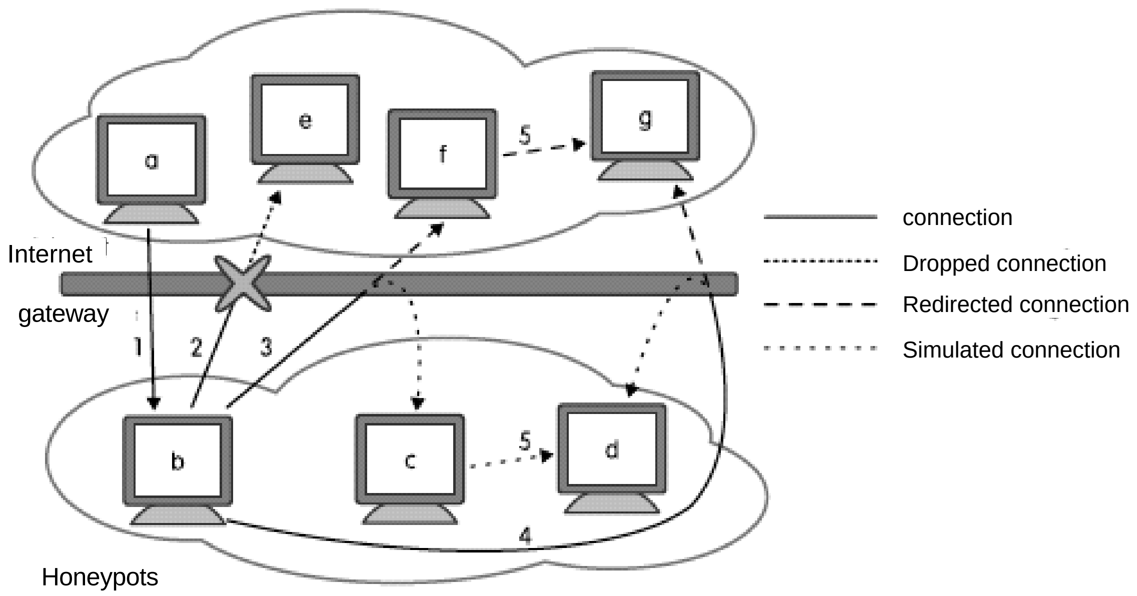
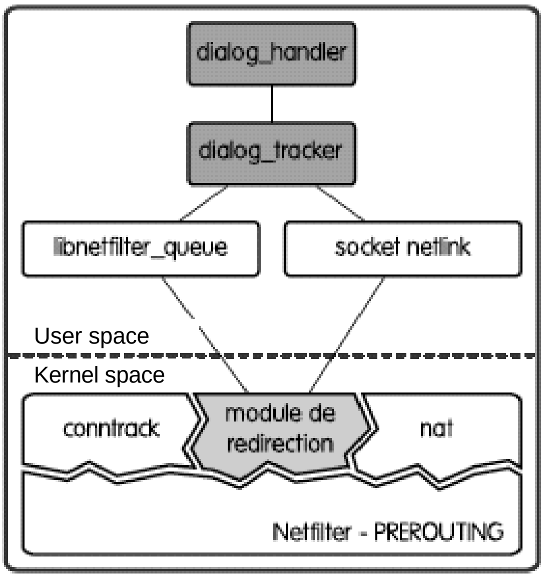


Figure 10- Connection redirection principles: example

The dynamic redirection mechanism has been implemented in the Gnu/Linux operating system through the `NETFILTER` firewall of the kernel. This firewall allows the interception and the modification of the packets flowing through the IP stack. As illustrated in Figure 9, the mechanism includes three components:

- the *redirection module* (inside the kernel) extracts the received packets.
- the *dialog_handler* decides whether the extracted packets must be redirected or not. Several algorithms can be used for this purpose and for the distribution of the redirected connections among the local honeypots.
- the *dialog_tracker* maintains the link between the redirection module and the dialog_handler. This way, the implementation of the dialog handler can be totally independent of the architecture and the operating system. In particular, the dialog_handler and the dialog_tracker could be run on different machines.

The implementation of the three components is described in detail in [Alata *et al.* 2007], with experimental results showing that the redirection mechanism does not lead to a significant latency due to the interception and redirection of connections. This is important to ensure that the overhead is sufficiently low as to prevent detection of the redirection mechanism by the attacker.



memory

Figure 11- Redirection mechanism implementation through NETFILTER

3.3.3 Deployment

In order to collect real data about attackers activities and to validate our set up, we have deployed our virtual high-interaction honeypot on the Internet. Figure 12 gives an overview of the most recent version of the honeypot. Three Gnu/Linux virtual machines M1, M2 and M3 have been set up. Each machine includes usual desktop software (Compiler, Text editors, etc.). Only M1 and M2 are accessible from the Internet. M3 is accessible from the other virtual machines. The only input connections authorized by the firewall are those targeting the ssh service. Concerning output connections from the honeypot, two virtual machines R1 and R2 are used by the redirection mechanism.

The deployment was carried out in two stages. The first deployed version did not include the redirection mechanism and the virtual machines have been set up using VMware. This setup was then upgraded using Qemu and included the redirection mechanism.

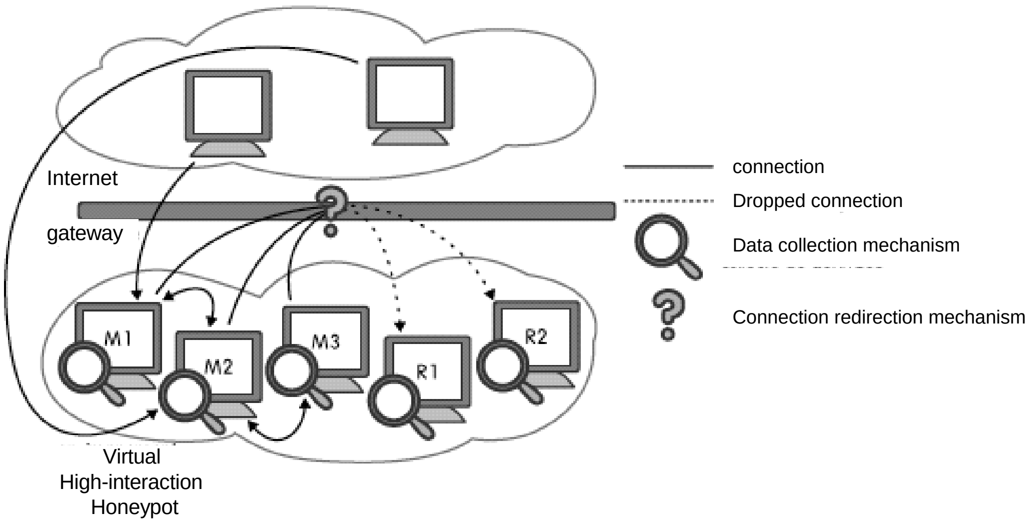

Internet
gateway
M1
M2
M3
R1
R2
Virtual High-interaction Honeypot
connection
Dropped connection
Data collection mechanism
Connection redirection mechanism

Figure 12- Overview of the most recently deployed version of the honeypot

In the beginning of the experiment (approximately one and a half month), we deployed a machine with a `ssh` server correctly configured, offering no weak account and password. We have taken advantage of this observation period to determine which accounts were mostly tried by automated scripts. Using this acquired knowledge, we have created 17 user accounts and we started looking for successful intrusions. Some of the created accounts were among the most attacked ones and others not. As we already explained in previous sections, we have deliberately created user accounts with weak passwords (except for the `root` account).

As illustrated on Figure 13, we can distinguish two main steps for the activities recorded for each user account, identified by the durations τ1 and τ2. The first one measures the duration between the creation of the account and the first successful connection to this account, and the second one measures the duration between the first the first successful connection and the first real intrusion (i.e., a successful connection with commands). Table 5 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 user account is successfully found and the time when someone logs into the system with this account to issue some commands on the now compromised host. This is a somehow a surprising fact. 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.

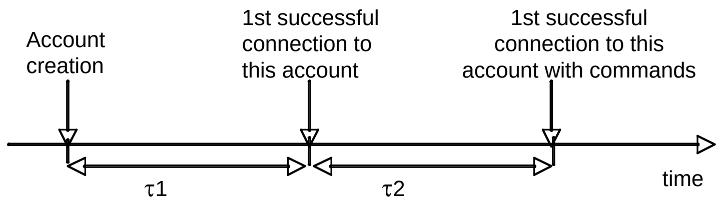

Account creation
1st successful connection to this account
1st successful connection to this account with commands
τ1
τ2
time

Figure 13: Definitions of τ1 and τ2

Table 5: τ1 and τ2 values for the each user account

User Account	τ1	τ2
UA1	1 day	4 days
UA2	1.5 day	4 minutes
UA3	15 days	1 day
UA4	5 days	10 days
UA5	5 days	0
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

As regards the data collected so far during the experiment, the number of `ssh` connection attempts to the honeypot that we have recorded is 552362 (we do not consider here the scans on the `ssh` port). This represents about 1318 connection attempts a day. Among these 552362 connection attempts, only 299 were successful. The total number of accounts tested is 98347 and the number of different IP addresses observed on the honeypot was 654. This represents a significantly large volume of data on which statistical analyses can be carried out to extract relevant information about the observed attack processes.

This analysis is currently undergoing and the results will be presented in the next year deliverable.

4 CONCLUSION

In this deliverable, we presented two complementary experimental environments that are aimed to support the activities carried out in CRUTIAL in order: 1) to identify security-related vulnerabilities in software components and servers used in the CRUTIAL architecture, and 2) to collect real data representative of attacks typically observed on the Internet that will be useful to build models characterizing malicious threats.

The identification of security-related vulnerabilities is based on the injection of attacks using a new tool (AJECT). The experimental results obtained so far have confirmed that the methodology developed and the features offered by the tool are very relevant and well suited to identify residual vulnerabilities in software components and servers, such as those used in the information infrastructures investigated in CRUTIAL. The work that will be carried in the next year will be focussed on the application of the tool to selected components of the CRUTIAL reference architecture considering two types of components: the CIS devices that are positioned at the borders of the protected networks and the servers that provide fundamentals services to the other components of the networks (e.g., the DNS servers).

Concerning the collection and analysis of attack data based on honeypots, the effort will be focussed on the processing and the analysis of the data collected during the deployment of our high-interaction honeypot on the Internet in order to characterize the observed scenarios and behaviours of the attackers. Additionally, we will work on the enhancement of the capabilities offered by the current implementation, by including additional vulnerabilities that can be used by potential attackers to compromise systems and servers connected to the Internet.

REFERENCES

[Abell 2007] V. A. Abell, “lsof – LiSt Open Files”, http://people.freebsd.org/~abe/, 2007.

[Alata et al. 2007] E. Alata, I. Alberdi, V. Nicomette, P. Owezarski and M. Kaâniche, “Internet Attacks Monitoring with Dynamic Connection Redirection Mechanisms”, Journal in Computer Virology 2007.

[Alberdi et al. 2005] I. Alberdi, J. Gabès and E. L. Jamtel, “Uberlogger : Un observatoire niveau noyau pour la lutte informatique défensive”, Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC-2005), Rennes, France, 2005.

[Amir et al. 2006] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, “Scaling byzantine fault-tolerant replication to wide area networks” in: Proc. Int. Conf. on Dependable Systems and Networks, pp.105-114, June 2006.

[Artail et al. 2006] H. Artail, H. Safa, M. Sraj, I. Kuwatly and Z. Al Masri, “A Hybrid honeypot framework for improving Intrusion detection Systems in protecting organizational networks”, Computers & Security, 25 (4), pp.274-288, 2006.

[Bailey et al. 2005] M. Bailey, E. Cooke, F. Jahanian and J. Nazario, “The Internet Motion Sensor - A Distributed Blackhole Monitoring System”, in Network and Distributed Systems Security Symposium (NDSS-2005), (San Diego, CA, USA), 2005.

[Baulig & Kacar 2007] M. Baulig and D. Kacar, “LibGTop -- Library that provides system information”, http://directory.fsf.org/libs/LibGTop.html, 2007.

[CAIDA] CAIDA, “CAIDA, The Cooperative Association for Internet Data Analysis, http://www.caida.org”.

[Crispin 2003] M. Crispin, 2003, Internet Message Access Protocol - Version 4rev1. Internet Engineering Task Force, RFC 3501.

[Cook et al. 2004] E. Cook, M. Bailey, Z. Morley Mao, D. Watson, F. Jahanian and D. McPherson, “Toward Understanding Distributed Blackhole placement”, in The 2004 ACM Workshop on Rapid Malcode (WORM'04), (New York), pp.54-64, ACM Press, 2004.

[DShield] DShield, “The SANS Institute. Distributed Intrusion Detection System, http://www.dshield.org.”

[van Eeten et al. 2006] M. van Eeten, E. Roe, P. Schulman, and M. de Bruijne, “The enemy within: System complexity and organizational surprises”, M. Dunn and V. Mauer, editors, Int. CIIP Handbook 2006, 2, pp.89-110, CSS, ETH Zurich, 2006.

[GNU Foundation 2006] GNU Fountation, 2006. GDB. Http://www.gnu.org/software/gdb/.

[ISC] ISC, “The SANS Institute. Internet Storm Center. http://isc.sans.org/”.

[Koziol et al. 2004] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell, “The Shellcoder’s Handbook: Discovering and Exploiting Security Holes”, John Wiley & Sons, 2004.

[London et al. 2001] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak, “The PAPI Cross-Platform Interface to Hardware Performance Counters”, in Department of Defense Users' Group Conference Proceedings, 2001.

[Moore et al. 2001] D. Moore, G. M. Voelker and S. Savage, “Inferring Internet Denial of Service Activity”, in 10th USENIX Security Symposium, 2001.

[Pax 2007] Pax, “The Pax Team”, (6 December 2007), 2007, http://pax.grsecurity.net.

[Pettersson 2002] M. Pettersson, “Linux Performance-Monitoring Counters Driver”, http://www.csd.uu.se/~mikpe/linux/perfctr, 2002.

[Pouget et al. 2005] F. Pouget, M. Dacier and V.-H. Pham, “Leurré.com: On the Advantages of Deploying a Large Scale Distributed Honeypot Platform”, E-Crime and Computer Conference (ECCE '05), Monaco, 2005.

[Powell & Stroud 2002] D. Powell, & R. Stroud, editors, “Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21”, http://www.research.ec.org/maftia/deliverables/D21.pdf, 2002.

[Provos 2004] N. Provos, “A virtual honeypot framework”, in 12th USENIX Security Symposium, pp.1-14, 2004.

[Provos & Holz 2007] N. Provos and T. Holz, Virtual Honeypots — From Botnet Tracking to Intrusion Detection, Addison-Wesley, Boston, MA, USA, 2007.

[Spitzner 2002] L. Spitzner, Honeypots: Tracking Hackers, Addison-Wesley, Boston, 2002.

[Vanderavero et al. 2004] N. Vanderavero, X. Brouckaert, O. Bonaventure and B. Le Charlier, “ The HoneyTank: a scalable approach to collect Malicious Internet traffic,” in International Infrastructure Survivability Workshop (IISW'04), (Lisbon, Portugal), 2004.

[Veríssimo et al. 2000] P. Veríssimo, N. F. Neves, and M. Correia, “The Middleware Architecture of MAFTIA: A Blueprint”, in: Proceedings of the Third IEEE Information Survivability Workshop, 2000.

[Veríssimo et al. 2003] P. Veríssimo, N. F. Neves, and M. Correia, “Intrusion-Tolerant Architectures: Concepts and Design”, in: R. Lemos, C. Gacek, & A. Romanovsky, editors, Architecting Dependable Systems, vol. 2677 of Lecture Notes in Computer Science, pp.3-36, 2003.

[Wilson 2006] C. Wilson, “Terrorist capabilities for cyber-attack”, M. Dunn and V. Mauer, editors, International CIIP Handbook 2006, 2, pp.69-88. CSS, ETH Zurich, 2006.