Using wrappers to protect specific network services

A Final Paper presented to the

Faculty of the School of Computer Science

Kennedy-Western University

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science in Computer Science

by

Gunnar Eyal Wolf Iszaevich

Mexico D.F., Mexico

February 28, 2001

Copyright and Licensing details

Copyright (c) 2000 Gunnar Wolf.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with the Front-Cover Texts being the original author's name (Gunnar Wolf), the original title (Using wrappers to protect specific network services), and the original intention of this paper (final paper for Computer Science at Kennedy-Western University), and with no Back-Cover Texts.

The full text of the both the GPL and the FDL licenses can be found at http://www.gnu.org.

Abstract

In this work the author addresses some of the currently most dangerous security issues for Internet servers which are responsible for most of the attacks suffered by servers every day: Protocol misuse and buffer overflows. Both problems can be addressed with a single approach, which will be explained and built in the following pages: Protocol-specific TCP service wrappers.

This project started with much less ambitious goals. Originally, it pursued making a tool to help system administrators fight spam (unsolicited bulk mail, see section 3.5.1 and reference [5, Sendmail 8.9.3 con Reglas Anti-Spam]), which has become one of the major headaches of the Internet users over the last few years. This is how the idea of a protocol-specific wrapper was born: A wrapper that would monitor and, if necessary, limit the client's interaction with the server, filtering out some potentially dangerous commands (described in section 3.5.2). However, after some weeks, the author realized that this work could greatly benefit many other services if it was done as a generic, customizable wrapper.

Having worked since November 1999 in UNAM's Computer Security Department[1]

[1] UNAM (Universidad Nacional Autónoma de México) is the largest and most important University in Mexico. Within DGSCA (Dirección General de Servicios de Cómputo Académico), its largest computer related division, lies the Computer Security Department, the most serious and important emergency response team and computer security research area in Mexico. Its URL is http://www.asc.unam.mx

and after over three years of being a security-conscious UNIX system administrator, the author found there was a great need for such a product. History has proved that most successful attacks manage to break into a system exploiting buffer overflows —sending more data than what a program expects in any given moment, exceeding the space designed to hold it and usually overwriting vital parts of the program's image in main memory and subverting it. This kind of attacks are usually not hard to patch at the source code level, but sometimes that is not enough. The author saw the need for a proactive defense scheme to save the administrators from the disable-download-patch-enable-pray cycle, building a wrapper immune to buffer overflows, or at least a wrapper which would not compromise the entire system if successfully attacked.

When comparing the programming languages with which to do this project, using Perl was chosen over C and C++ because, albeit being a slower, non-compiled language, it has many desirable characteristics, notoriously the ability to manage the memory without exposing the programmer to make a critical mistake in such a critical part, and powerful pattern recognition techniques.

It should be noted that this work is a proof of concept; a working implementation of the suggested ideas, but it should be viewed more as an invitation for other programmers to continue working on this program or even reimplement it if a better way to do so is found than as a finished product.

Contents

1. Introduction **1**

1.1. Statement of the problem 2

1.2. Historical perspective 2

1.3. Purpose of study 6

1.4. Importance of study 7

1.5. Scope of study 7

1.6. Rationale of study 8

1.7. Definition of terms 9

1.8. Overview of study 11

1.9. Copyright and licensing 12

2. Review of related literature 13

2.1. The Perl language 13

2.1.1. Reasons for choosing Perl 14

2.1.2. Reasons why Perl might not be an ideal choice 16

2.2. Program implementation 17

2.2.1. Program requirements 17

2.2.2. When not to use a wrapper 20

2.3. RFC analysis 20

2.3.1. SMTP - RFC 821 21

2.3.2. POP3 - RFC 1081 26

3. The ProtoWrap wrapper implementation 32

3.1. Approach followed for this work 33

3.2. Database of study 34

3.3. Basic architecture 35

3.4. Interaction with the base ProtoWrap wrapper 38

3.5. Protocol-specific extensions: SMTP 42
3.5.1. The spamming problem 42
3.5.2. Characteristics of a SMTP server 48
3.5.3. Implementation details 51
3.6. Protocol-specific extensions: POP3 51
3.6.1. Characteristics of a POP3 server 52
3.6.2. Implementation details 56
3.7. Protocols deliberately not contemplated 56
3.7.1. HTTP 57
3.7.2. FTP 57
3.7.3. Telnet 58
3.7.4. SSH 59
3.7.5. UDP and ICMP based protocols 59
3.8. SSL encrypted protocols 60
3.9. Validity of data 61
3.10. Originality and limitations of data 62
3.10.1. Originality 62

3.10.2. Limitations 63

3.11. Summary 64

4. Implementation **66**

4.1. The dual I/O problem 66

4.2. Dealing with different data sources 68

4.3. Reimplementing server functionality 71

4.3.1. Why must functionality be reimplemented 72

4.3.2. Matching the server's lost functionality 74

4.3.3. Enhancing functionality 75

4.4. Excessive logging and privacy concerns 77

4.5. Attaining low ports with low privilege 79

4.6. Auto-looping 80

5. Conclusions **83**

5.1. Future enhancements to ProtoWrap 83

5.1.1. Master initialization program and configuration file 84

5.1.2. Signal handling 85

5.1.3. Load balancing 87

5.1.4. Perl-style (POD) documentation 91

5.2. Sample configurations 92

5.2.1. Wrapper running at the firewall 93

5.2.2. Redirecting firewall 95

5.2.3. Server not running, called by the wrapper 98

5.2.4. Server running on a different port 101

5.2.5. Local redirecting firewall local rules 104

5.2.6. Other configurations 106

A. ProtoWrap program code **107**

A.1. Makefile.PL 107

A.2. ProtoWrap.pm 107

A.3. ProtoWrap/SMTP.pm 117

A.4. ProtoWrap/POP3.pm 126

1. Introduction

This chapter describes the steps that led the author to develop ProtoWrap.

Section 1.2 starts with the historical perspective of Internet and a brief description of the most common of attacks against the servers, together with some approaches that have been taken to prevent them, as well as the author's proposed approach.

Sections 1.1, 1.3, 1.4, 1.5, 1.6, 1.7 and 1.8 formally state the work's mission, importance and rationale.

Section 1.9 states the copyright and licensing terms under which this work and the code resulting from it are protected.

1.1. Statement of the problem

Today, Internet is not the safe network it once was. The Internet is full of hostile elements, trying to find a way to attack and gain control of our system for various purposes. Most successful attacks exploit well-known weaknesses derived from insecure programming techniques, specially buffer overflows and protocol abuse. No program can be expected to be completely secure, however how carefully it is written.

1.2. Historical perspective

The Internet is a very large, very important communications network. It is synonymous with modernity and technology. However, its design starts showing its age in several aspects.

Internet was built as a cooperative, trusting network - Every node makes its best to facilitate communication between any two other nodes than might request it, and trusts their requests to be legitimate. This design comes from the first days of the network —then, the *Arpanet*— when its military origins led to research on how to make a network that would keep working even if part of it was damaged.

Originating in the research and military arenas, the Internet we know today in-

herited a very positive and open structure: A vast majority of the implemented protocols are based on open standards, and are thoroughly documented in the *IETF RFC*s (Internet Engineering Task Force Request For Comments). Knowing that is the highest source of authority on Internet standards, the author decided to base most of his research on their texts.

The Internet relies mainly in Unix servers for its operation - Unix is an operating system with a long history, with over 30 years in use, and it has proven ideal for the job. The actual suite of protocols that make Internet exist (TCP/IP, UDP/IP, ICMP/IP) were designed on Unix systems, and they still excel at its efficient and compatible implementations.

A phenomenon not new to us is that of external attackers (*crackers*), people who use vulnerabilities or errors in a given system to exploit it, either to get information restricted to people with authorized access, to modify information or to stop the computer from serving requests. Answers to these attacks have been broad, even overwhelming:

The first and most obvious path followed was bug-fixing: As soon as a bug is detected, people with enough understanding of the attacked program search for the error in the code and fix it. This technique, although the most efficient, has major shortcomings: How long will it take the programmers to identify the error? How long will it take to all the users of the program get the updated software and

install it? Meanwhile, how many other systems will be attacked? If the program is commercial, as most software is nowadays, will the vendor acknowledge it had a problem, or will he just pretend nothing happened, until the next version is shipped with this bug patched? Some of those shortcomings are almost nonexistent with free software, such as the FreeBSD, NetBSD, OpenBSD and Linux operating systems, because of the large number of technical users who know the deepest intricacies of their programs, but even there, they can not be overlooked. System administrators are still required to manually patch and update the programs, and this is the most overlooked aspect of security. Administrators are confident in their system's configuration, albeit they still run outdated and unmantained versions of vital components of the system.

A second, pro-active path is the one followed by the OpenBSD team: Secure code auditing[1]. The OpenBSD team took the whole NetBSD free operating system, and began auditing it, line by line, program by program, searching for wrong or potentially dangerous programming or operating techniques. After a tremendous amount of work, they produced the most secure operating system to date. Of course, this highly effective technique also has its shortcomings: New services and new, updated versions of the programs have longer delays to show up on their system because they must be audited, and priority is given to keep auditing the existing code. Thus, OpenBSD is an ideal system for a firewall or for a server

exposed to frequent attacks, but it may not be adequate for a normal server, and is usually outright discarded for a personal workstation.

A third approach is the use of firewalls: intermediate systems dedicated to filtering out specific ports or suspicious activity on the external network in order to avoid dangers to the organization's servers or data. Although firewalls are not hot news anymore and most medium and big installations have at least a basic one, they often fall short of our current needs, doing all their work on a source-destination address and port checking. Only a handful of them are able to do stateful inspection [1]of the packages, and they still fail to stop or even detect most basic attacks.

Yet another approach is the hardening not only of the programs, but of the compiler. Stackguard [2], the key part of the Immunix project, is a compiler that warns the programmer about possible insecure programming techniques, and produces binaries that are much more resistant against stack overflows. Of course, this has the disadvantage that the whole system has to be recompiled, making it impossible to be implemented on propietary systems, such as Windows or Solaris, and viable only for free systems, such as Linux and the *BSDs. Further, the whole system has to be recompiled in order to get a trustable installation —a monstrous task. Some

[1] Analyzing each incoming packet, working not only on source and destination of the packages, but keeping track on whether the connection was originated by which party (often allowing outbound connections while rejecting inbound connections) and the packet sequence (avoiding, to a certain degree, connection hijacking).

programs will not compile correctly with a compiler that is much stricter than standard ANSI C compilers, and routines avoiding stack overflows may make the whole operation of the system slower. Immunix addresses the recompilation problematic by supplying a ready-to-use distribution for Intel-based computers based on RedHat Linux 6.2. However, installing any package distributed in binary —and very likely not compiled with Stackguard— will open a possible security breach.

1.3. Purpose of study

System administrators lack a tool to help them spot potential attacks and stop them before it is too late and the intruder has penetrated our system. This work studies the development of an extra layer —a wrapper— between the client and the server, which becomes a natural extension to a firewall, checking syntax and validity on each line sent to the server. Here, we do not stop at checking source and destination, but we have a system acting as an intermediary between the remote host and the server, checking each line for syntax and validity - as well as some other rules we might want to add to a specific service.

1.4. Importance of study

Each day Internet is more and more important to us — It has penetrated almost every aspect of our day to day life, including personal communications, entertainment, shopping, business, etc. We need, therefore, to be able to trust the servers, to be sure they are reliable. Every step taken to make our servers more reliable is a step in the right direction. Every obstacle we put in the intruders' way will be a valuable addition to our servers' overall security.

This work is not unique in its approach. Different programs, such as TCP Wrappers, by Wietse Venema [3] or SMAP, by Marcus Ranum [4], have been written with the idea of wrapping around protocols. None, however, has been written in the spirit presented in this work: To be both a multiple purpose wrapper, protecting different servers and protocols, and to be an informed wrapper, examining each line before it reaches the server.

1.5. Scope of study

All work has been done on Unix systems with TCP/IP Internet connectivity, as they represent the bulk of the Internet connected servers.

The resulting program will be able to wrap most line-oriented TCP/IP protocols.

Not every protocol will be supported - for more details on which protocols can and can not be supported, refer to section 3.7.

The program resulting from this work is written in Perl (for more information on why Perl was chosen, see section 2.1), so probably they will run without significant changes —or without any changes at all— in other operating systems in which Perl can be run, such as Windows or MacOS.[2]

1.6. Rationale of study

The author shows with this work how, by building a wrapper that will act as an intermediary between the client and server programs, different problems can be avoided. These problems include:

- Buffer overflow
- Protocol abuse/misuse
- Information gathering
- Spam

[2]MacOS X is a full-fledged BSD-derived Unix system, and users are encouraged to take advantage of that fact. Although MacPerl is available for older MacOS versions, it is not reccomended anymore because of the lacking functionality the old MacOS enviroment imposed on its development; regular Perl should be used instead.

The author also discusses examples where a wrapper should not be used, and the reasons not to use it.

1.7. Definition of terms

Attack Any attempt to either obtain higher than normal access or to intentionally deny other users proper use of a computer facility

Buffer overflow When a program recieves more data than what it expected and is not programmed to cope with such situations, the extra data will be put in memory, probably overwriting other internal structures of a program. A well-crafted buffer overflow can lead the attacked program to execute arbitrary code, possibly giving the attacker complete control over the computer.

DoS (Denial of Service) An attack or situation where the victim host is presented with a much larger amount of request than what it can process, effectively blocking the service to any user who legitimately needs to use it.

Firewall A computer that sits at the perimeter of an internal, trusted network, connected to a larger, less-trusted network (such as Internt), filtering and recording undesirable packets from/to the outside world in order to enhance security for the internal network.

IETF RFC Internet Engineering Task Force's Request For Comments are a set of documents discussing and defining network protocols and other standards required for the correct inter-network operation needed for today's Internet to function.

Line-oriented A protocol that sets the boundary for any client or server command to be the new line character.[3]

Protocol A set of rules defining how a specific task should be handled by the computer. All computer network communications are carried out through several layers of protocols, ranging from the physical, concrete layer (electric signals and their immediate meanings) to much more abstract layers.

RFC *see IETF RFC*

root The privileged super-user account, used for administration purposes. Attackers often are looking for a way to forcibly get access to the root account, as there is practically no limit to what it can do in a Unix system.

TCP/IP Transmission Control Protocol/Internet Protocol, a protocol for Internet applications which require session handling or delivery acknowledgement,

[3]For most TCP/IP protocols, the new line character is a carriage return followed by a line feed, also written as CR-LF or \r\n

such as FTP, Telnet, SMTP, SSH and many others. This work is centered on line-oriented TCP/IP-based protocols.

UDP/IP User Datagram Protocol/Internet Protocol, a stateless protocol for Internet applications which do not require session handling nor delivery acknowledgement, such as DNS, BootP, TFTP, RPC and many others.

UID User ID, a number identifying each user in a Unix system. UID 0 is reserved for the root user.

Unix The prevalent server operating system in use in Internet; a multi-tasking, multi-user operating system designed in 1969 by Bell Labs

1.8. Overview of study

The work was done in three main phases:

Research and planning Research about similar projects and the protocols proposed to be covered.

Engine Selecting the language to be used, the type and depth of analysis to be performed, and programming the main program engine,

Extensions Choosing the protocols to be covered in this work, the protocols *not* to be covered, analyzing the RFCs for the chosen protocols, and programming the specific engines.

1.9. Copyright and licensing

This document is covered by the GNU FDL (*Free Document License*), ensuring that it will always be freely available for anyone to learn, use, modify and adapt to any project he/she finds fit, complete or in part, modified or not modified.

The program resulting from this work is covered by the GNU GPL (*General Public License*), in order to guarantee its free distribution and development, and supporting it as the only viable license for academic works. According to the GNU GPL, the program will be free to use, distribute, learn from, modify, enhance, embed or use in any other concievable way, as long as it remains bound to this license.

The full text of the both the GPL and the FDL licenses can be found at http://www.gnu.org.

2. Review of related literature

The author discusses in this chapter the background on which he based his work.

The choice of the Perl language is discussed in section 2.1; section 2.2 describes what is a wrapper, what does it need to run and different strategies to run it and talks about the potential risks and pitfalls of using a wrapper. Finally, in section 2.3 the pertinent RFCs —the documents that gave birth to most Internet protocols— are analyzed.

2.1. The Perl language

When facing a project such as this one, the very first issue that should come to mind was the language to be used to implement it. Choosing Perl for this

project was almost straightforward. [10, Learning Perl], [11, Programming Perl] and [12, Advanced Perl Programming] are the best reference to date related to the Perl programming language, and are strongly reccomended to anyone seriously interested in learning Perl.

2.1.1. Reasons for choosing Perl

Perl is a free[1] programming language, which can be used as a solution for small problems with very little code, but which can also be extended to very ambitious projects.[2] Perl is one of the few languages which can be, according to the programmer's needs, non-structured, structured or object oriented —most languages provide for only one of those styles. For example, it is impossible to write structured code using Java, and objects are non-existant in C.

One of the most popular ways of cracking into a system is by abusing the network daemons with buffer overflows. Most C programs are at different points vulnerable to this kind of attacks: The attacker sends an unusually large amount of data to the server in a single request, exceeding the predefined space allocated to receive

[1] Free as in freedom, not only price. The author reccomends always using free software, for it is almost always more thoroughly tested, secure and efficient than any propietary counterpart. Choosing free software also allows any people interested in contributing to the project to do so with the least hassle possible.

[2] Quoting Larry Wall, the inventor of Perl: *Perl makes easy things easy, and difficult things possible.*

the request, and possibly sneaking into a space meant for code, thus forcing the victim to execute arbitrary code. The effects of such attacks can range from a relatively innocuous crash to an intruder gaining control of our system. Using a wrapper we can completely avoid buffer overflow attacks by limiting the amount of data that reaches the real server. However, in order to do so, the wrapper needs to be extremely careful not to experience a buffer overflow itself — falling back to the original problem. The author found Perl to be an ideal language for this, because memory is handled by the Perl compiler itself, not by the programmer (as opposed to C, where dynamically allocated memory and fixed size arrays can and very often do respectively lead to memory leaks and buffer overflows).

To implement wrappers, we need a language that makes it easy —optimally, transparent— to read from and write to the network. Perl do so, thanks to their `IO::Socket` module. The network can be read from and written to as if it were a code that is file or with the `getline` and `print` methods if we prefer using an object-oriented approach.

Having an object-oriented language is definitively desirable. In that way, a base wrapper class can be build and later extended with protocol-specific derived classes —of course, leaving it open for future users to implement any protocol they want to, without having to re-implement or modify the base class. Since version 5, Perl can be programmed in an object-oriented way.

2.1.2. Reasons why Perl might not be an ideal choice

While Perl is a very versatile language, it is not perfect for all tasks. First of all, the program must be compiled every time it is invoked. While the Perl compiler is very efficient, being able to compile very large programs in fractions of a second, it is a step that must be repeated many times, maybe innecesarily[3]. This makes the wrapper not suitable to be run as an inetd process if it will be called at least once every five seconds.

Perl is also not very conservative in RAM. Even the simplest program will take an absolute minimum of 1.5 Megabytes. This makes it impractical to run many instances of the wrapper as standalone processes on a limited machine. The author would advises having at least 5MB of available physical RAM for each wrapper class. ProtoWrap, in its present form, is not made to endure heavy load situations - in fact, a DoS attack can easily stop the machine from responding to requests[4] . It is however preferable for a computer to get trapped in a DoS attack, which

[3] There are several ways of compiling Perl code. However, as of Perl version 5.005, they are still regarded as *experimental*. With the recent announcement of Perl version 5.6 this may change, but until the new version is thoroughly tested and has a wide user base, the author decided to stick with 5.005's limitations.

[4] A computer suffering a *Denial of Service* (DoS) attack means it is being purposefully presented with many more requests than it can handle at a time, thus overloading it and forcing it to deny service to legitimate users. Using technology available today, a DoS attack is very difficult to detect, and even harder to stop.
In March 2000 the first reports of *Distributed Denial of Service* (DDoS) surfaced. They refer to a variety of DoS attack which is not carried out by a single computer, but by many simultaneous, being much stronger than any previously reported DoS.

poses no real danger to the configuration and data stored in the server, than to suffer direct attacks to the real daemons.

2.2. Program implementation

It is of fundamental importance for this work to understand, before going into greater detail, the basic points of the program implementation. The chosen license, the basic requirements and mechanisms to be able to run ProtoWrap and some comments on when using Protowrap can not only be undesirable but even harmful.

2.2.1. Program requirements

This wrapper implementation, called ProtoWrap is meant to run with Unix or Unix-like systems.[5] This is mainly because of their robustness and stability, specially when compared to other popular operating systems, and their adherence to open standards. This implementation was programmed and tested on Linux machines, but porting it to other Unix platforms should prove a trivial task.

[5] Legally, Linux, FreeBSD, NetBSD and OpenBSD are not Unix systems, even though they behave as if they were. This is because Unix is a registered trademark of Bell Labs, and royalties have not been paid to use the Unix name. They do, however, adhere to the POSIX standard which defines Unix systems. Thus, they are referred to as *Unix-like*, but anything regarding Unix applies also to these operating systems.

The program will obviously have to deal constantly with *sockets* - network services. While designing ProtoWrap, we faced two ways of opening a server process —a *daemon* in Unix jargon— and two ways of communicating with a client process depending on our needs.

As for the communication with the client, for services that are seldom called and do not have a big initialization overhead (either in time or in system resources), the best strategy is to call them from the *inetd super-daemon*, a program that listens on every port and, when an incoming connection requires it, calls the pre-defined program that should take care of that particular service. Services whose daemons are typically called from inetd usually include telnet, ftp, pop3, finger and other such protocols. When calling the specific program for the service type requested, inetd redirects its *stdin* and *stdout* to the already opened socket. The second strategy —the *standalone* mode— is to have the daemon always active. This is the case with heavily used services, such as http, or with services that require performing lengthy initializations, as the strong random number generation performed by ssh[6]. Such programs must be always alive, listening at their ports

[6] ssh —*secure shell*— is a relatively recently introduced protocol designed to replace and enhance telnet and the r-commands (rlogin, rcp, rsh). They require replacement because they are protocols which transmit information in *clear text*, applying no encryption and thus allowing anyone connected to the same network as the host or the client to intercept all of the information, or even —with relatively trivial cracking utilities— to hijack the connection. ssh also enhances telnet and rlogin by providing encrypted tunnels by which other protocols —usually, X-Window sessions— can be routed, while encrypting also their information. It includes also scp, designed to allow encrypted file transfers between computers.

and waiting for a connection. Daemons, however, use constantly a portion of the RAM and CPU cycles of the system, besides occupying a slot in its process table.

As for communicating to a server, a similar choice is presented. The server may be a program executable from the local machine, handling communication with its client via the standard input and output (*stdin* and *stdout*) file descriptors, or it can be executing at an arbitrary location —including the same computer— and communicating via a TCP/IP socket. The first strategy should be chosen if the server can be called from inetd, as it will improve performance not having to go twice through a TCP/IP stack, and security, not being able to be called without going through the wrapper or having local access to the computer. The second strategy should be used if ProtoWrap will work from a firewall or if the service being wrapped can not use *stdin/stdout* for its communication. Of course, if the second strategy has to be chosen, the system administrator can still limit —using *ipchains* or an equivalent program— the connectionsto the server to a specific computer.

ProtoWrap is compatible with all those strategies. Of course, when deciding which one to use, the user must evaluate how his system will be used, do a maximum expected load calculation, and not discard switching to other strategies should the usage patterns of the server change.

2.2.2. When not to use a wrapper

Although they will help us out in many situations, wrappers should not be always used. First and foremost, there are many protocols not very well suited for being wrapped, at least not with the techniques described in this work. Some of the most representative are listed in section 3.7.

Although this point has already been mentioned, it demands being repeated as much as possible: Very heavily loaded systems, as well as systems where high availability is the top priority and can not afford facing a DoS attack, are particularly bad candidates for using wrappers. Wrappers add extra work to the CPU and extra load to the system RAM, and in some cases this will cause more harm than good.

2.3. RFC analysis

As stated in the introduction, Internet protocols are based on open standards, and this standards are defined by the RFCs of the IETF.[7] In chapter ?? protocol-specific extensions to ProtoWrap will be defined — and in order to do so, a thorough understanding of the protocols is needed. In this section, the pertinent RFCs will be reviewed.

[7] RFC stands for Request For Comments, IETF stands for Internet Engineering Task Force

2.3.1. SMTP - RFC 821

The SMTP protocol has two different uses: Delivering mail from the user's mail program to his server and delivering mail from this mail server to the server which stores the destination user's account. Both actions, different as they may seem, handle network communication in the exact same way: The client[8] requests a connection to the server[9] using TCP port 25. Once the connection is established, the server expects a sequence of commands (defined by RFC 821[6]), and, if its configuration does not instruct him to do otherwise, accepts and delivers the requested messages.

SMTP is a line-oriented protocol. Requests are sent by the client as a text line (or block of text lines, in the `DATA` part), and their answers are returned to the client with a status number and, optionally, an explanation. The status number is 200-299 for successful operations, 300-399 to indicate further input is expected, 400-499 to signal failure to execute a request and 500-599 to report an error in the request.

[8] Client is defined as the computer requesting a service, regardless of its formal role
[9] A server, likewise, is the computer attending the request from a client

RFC 821 Commands

The three most important commands are `MAIL`, `RCPT` and `DATA`. RFC 821[6] defines them as follows:

> The first step in the procedure is the `MAIL` command. The <reverse-path> contains the source mailbox.
>
> `MAIL <SP> FROM:<reverse-path> <CRLF>`
>
> This command tells the SMTP-receiver that a new mail transaction is starting and to reset all its state tables and buffers, including any recipients or mail data. It gives the reverse-path which can be used to report errors. If accepted, the receiver-SMTP returns a 250 OK reply.
>
> The <reverse-path> can contain more than just a mailbox. The <reverse-path> is a reverse source routing list of hosts and source mailbox. The first host in the <reverse-path> should be the host sending this command.
>
> The second step in the procedure is the `RCPT` command.
>
> `RCPT <SP> TO:<forward-path> <CRLF>`
>
> This command gives a forward-path identifying one recipient. If accepted, the receiver-SMTP returns a 250 OK reply, and stores the

forward-path. If the recipient is unknown the receiver-SMTP returns a 550 Failure reply. This second step of the procedure can be repeated any number of times.

The <forward-path> can contain more than just a mailbox. The <forward-path> is a source routing list of hosts and the destination mailbox. The first host in the <forward-path> should be the host receiving this command.

The third step in the procedure is the `DATA` command.

```
DATA <CRLF>
```

If accepted, the receiver-SMTP returns a 354 Intermediate reply and considers all succeeding lines to be the message text. When the end of text is received and stored the SMTP-receiver sends a 250 OK reply.

Since the mail data is sent on the transmission channel the end of the mail data must be indicated so that the command and reply dialog can be resumed. SMTP indicates the end of the mail data by sending a line containing only a period. A transparency procedure is used to prevent this from interfering with the user's text.

Please note that the mail data includes the memo header items such as Date, Subject, To, Cc, From.

> The end of mail data indicator also confirms the mail transaction and tells the receiver-SMTP to now process the stored recipients and mail data. If accepted, the receiver-SMTP returns a 250 OK reply. The `DATA` command should fail only if the mail transaction was incomplete (for example, no recipients), or if resources are not available.

While with these three commands any mail can be sent, the protocol contemplates many more commands, some of them deprecated, and some are even considered security threats:

- `VRFY` (verify) asks the server if a given address exists at that computer, answering with the user's complete name.
- `EXPN` (expand) is similar to `VRFY` in that it checks if a given address is valid, but if the address is a mailing list or a user's `.forward` entry, it answers with the names and addresses of all the recipients.
- `HELO` (opening session) is a polite way of telling the server who is connecting. Although this is no longer needed today that SMTP travels almost exclusively over TCP/IP, which gives this information to the daemon, it is almost always implemented for the sake of politeness.[10]

[10] Although nowadays SMTP is practically only used over a TCP/IP protocol stack, RFC 821 contemplates other possible network protocols, such as the Arpanet's NCP (*Network Control Protocol*), X.25 and NITS (*Network Independent Transport Service*).

- `QUIT` (closing session) tells the server the transaction is over and the connection may be terminated.
- `RSET` (reset, abort current commands) is used to abort an ongoing transaction, either to correct the given data or to abort and quit nicely without leaving an open session behind.
- `SEND` (send), `SOML` (Send Or Mail) and `SAML` (Send And Mail) are deprecated. They are used to send a mail message to the user's console. Nowadays, most users instead of consoles use graphical mail programs, and they rarely log in to the server, so this three commands are of no use.
- `TURN` (reverting roles) inverts the roles in a communication - The server becomes a client and the client becomes a server. This is very rarely used, and many SMTP implementations have opted not to implement it.
- `HELP` (help on SMTP use) is also considered deprecated because users do not usually talk to the SMTP servers anymore, leaving that task to the back-ends of user-friendly mail programs which have already all the needed knowledge to operate, so help is no longer needed.
- `NOOP` (No operation) does nothing, it is useful only for getting a positive answer from the server. It is also very seldom used, if at all.

2.3.2. POP3 - RFC 1081

According to RFC 1081, POP3, which is a successor to the Post Office Protocol described in RFC 918 (J.K. Reynolds, Oct-01-1984) and to the Post Office Protocol version 2 described in RFC 937 (M. Butler, J. Postel, D. Chase, J. Goldberger, J.K. Reynolds, Feb-01-1985), was designed with small systems in mind. What is a small system? A computer which is not connected continuously to the Internet or a computer which lacks enough resources (speed, RAM, bandwidth and disk space are the most common constrainers) to be a full-fledged multiuser mail server - a *Message Transport System*, or MTS. POP3 implements the necessary commands to allow the user of such machine to poll a larger computer at the intervals the user sees as most convenient to get the mail messages stored at the server and transfer them to his personal computer (usually, but not always, deleting them from the server). This computer needs only to be able to run a *User Agent*, or UA.

Nowadays, most mail is handled this way. Very few Internet users have their computer online all the time, and even fewer have the knowledge required to set up and administer a mail server. This makes POP3 one of the most important protocols existing today.

POP3 is, as SMTP, a very straightforward, line-oriented protocol. The client will

open an arbitrary port and request a connection to the server's TCP port 110.

Result codes for each command are handled in the most simple way possible: If a command issued by the client succeeds, a line starting with +OK (optionally followed by a text string) is sent back. If the command fails, a line starting with -ERR (again, optionally followed by a text string) is sent back.

If a multi-line answer is to be sent (only possible with certain commands), the server will indicate the answer is over with the sequence <CR><LF>.<CR><LF> (a period on an empty line).

RFC 1081 Commands

As soon a connection is established, the client will be greeted by a success string, usually containing the name and version of the server software, and it enters the first stage of a POP3 connection: the AUTHORIZATION state.

In this first state, there are only three possible commands:

- USER followed by a username tells the server who is trying to log in to get his messages. Most servers answer with a success string, regardless of the existence of the username, in order not to unwillingly give a possible attacker the list of valid usernames existing in the server.

- `PASS` followed by the user's password, gives the server enough information to identify the user. If the password matches the username and the user's mailbox is currently readable and lockable[11], a success string is sent to the client, usually containing the number of pending messages, and the session proceeds to the TRANSACTION. If the password does not match the username or the user's mailbox can not be locked, a failure string is sent back, explaining the reason for failure, and remains in the AUTHORIZATION state.

- `QUIT` tells the server to drop the connection. It always gets a successful answer.

The TRANSACTION state implements the following commands:

- `STAT` returns a successful answer, followed by a space, the number of pending messages, a space and the total number of bytes waiting to be sent. The RFC strongly suggests that the server should add no comments to this answer, in order to make it easier to be parsed:

  C: `STAT`

  S: `+OK 3 5710`

[11] Every serious multiuser operating system will allow for a method of telling if a file is currently in use and denying write access to it, in order to preserve the file's integrity.

- `LIST` with no arguments returns a list of messages in the a format like the following:

  C: `LIST`

  S: `+OK 3 messages (5710 octets)`

  S: `1 1206`

  S: `2 2001`

  S: `3 2503`

  S: `.`

  Which means, the message number and the number of bytes it has.

  If LIST is called with a numeric argument, it answers with a format similar to `STAT`, indicating the message number and its length, or a failure string if the requested message does not exist or has been deleted.

- `RETR` requires a numeric argument, and returns the whole text of the requested message. If the message is deleted or non-existent the answer will be a failure string. Otherwise, it will be in the following format:

  C: `RETR 1`

  S: `+OK Message follows`

  S: <the entire message>

  S: `.`

  If the number given is higher than the highest message number accessed,

the pointer to the highest message accessed is set to one past the deleted message number.

- `DELE` takes a numeric argument, and marks the corresponding message as deleted. If the message does not exist or it has already been marked as deleted, a failure string will be returned. Otherwise, a success string will be returned. If the number given is higher than the highest message accessed, the pointer to the highest message number accessed is set to one past the deleted message number. Note that the message is not actually deleted until the TRANSACTION state is finished.

- `NOOP` returns a success string, and does nothing.

- `LAST` returns a success string and the last message number accessed, or zero if no messages have been accessed.

- `RSET` returns a success string, and is the equivalent of starting over the TRANSACTION state: It clears all the deleted flags set by `DELE` and resets the last message number accessed to zero.

- `QUIT` leaves the TRANSACTION state and enters the UPDATE state. When leaving the transaction state with this command, the messages marked to be deleted are effectively purged off the mailbox.

The following optional commands are also defined. The RFC states they can be left out of a particular implementation:

- `TOP` (TRANSACTION state) takes two numeric arguments: The message number and the number of lines. If the requested message exists, the server returns a success string, followed by the message headers and the indicated number of lines of the message's text.
- `RPOP` (AUTHORIZATION state) performs a function similar to `PASS`, although with a different mechanism: It authenticates a user based on whether it comes from a trusted host[12].

[12]defined with the .rhosts file in the user's home directory or with the hosts.equiv file in the server's /etc directory. This is not reccomended in open or large networks, as a simple IP spoofing may very easlily compromise the systems' security

3. The ProtoWrap wrapper implementation

This chapter inspects the inner design of ProtoWrap. In section 3.1 the approach followed for this work is outlined. Section 3.3 details how the wrapper is called and which attributes does ProtoWrap provide. Section 3.4 explains the methods ProtoWrap objects have and how to start, stop and query a wrapper.

Sections 3.5 discusses in depth the SMTP protocol, whose many problems motivated this work, and the suggested implementation covering most of its shortcomings. This section, together with 3.6, shows how to extend ProtoWrap to cover additional protocols. In section 3.7 many other protocols are reviewed, explaining the reasons they are not fit for wrapping with ProtoWrapper. Section 3.8 makes a

quick note on different ways to achieve SSL encryption for communications, and comments on wrapping SSL encrypted protocols.

Sections 3.9 and 3.10 formalize on the validity, originality and limitations of the data.

In section 3.11 the author presents a brief summary of this chapter.

The fully commented code relevant to this chapter is located in appendix A.

3.1. Approach followed for this work

This work evolved from a much simpler concept into what it now is, a very ambitious, multiple-purpose security system. The original goal of this work was to protect mail server from spam and similar forms of abuse. After studying several different solutions available ([4], [5]) and finding them lacking in functions or too difficult for the average user to implement, work started on a SMTP-specific wrapper designed to watch the SMTP commands sent to the server, and to disallow messages matching a certain criteria.

When the author started coding, it became obvious that, working a little bit more, the same work could be used with much broader applications and help in a much larger number of ways. The project broadened its focus, becoming a generic wrapper, able to be extended to protect different wrappers.

After an initial documentation phase, attention was paid to writing the core code, with the minimum possible of functions to be useful by itself, but to be easily expandable. Here, the author must wholeheartedly thank Salvador Ortiz García for his help with the dual I/O problem (discussed in section 4.1).

After having a working core, an exhaustive analysis of the RFCs for several protocols —most importantly, SMTP and POP3, the two protocols this work is based on— was done, leading to the protocol-specific code.

Finally, having the code ready for use, it was tested on the author's servers, fixing the bugs that appeared at runtime.

Further analysis and revision of the code will always be important. As stated in [1], code must be thoroughly audited by experts to be considered safe. Once again, this work should only be considered a proof of concept, not a finished product. It is not advised to be run as is on a production environment.

3.2. Database of study

This work was based on the study of different line-oriented TCP/IP protocol RFCs, as they are the highest authority on protocol implementation.

For the implementation, ProtoWrap was tested on the servers located at UNAM ENEP Iztacala, the author's workplace. It was tested using the Linux/i386,

Linux/Alpha and OpenBSD/i386 operating systems. As Perl is a highly portable language, the program should run with no modifications on almost any Unix system.

Due to the differences in the handling of file descriptors, it is highly improbable that it can run on Windows or MacOS (version 9 or lower) operating systems.

3.3. Basic architecture

In order to allow us to grow to a potentially complex system of wrappers, we need the basic ProtoWrap wrapper to be easily extendable, have an interface as simple as possible to understand, deal with and extend, and only to implement the most basic rules upon which all other rules will be based.

The base ProtoWrap wrapper is implemented as an object class, so that future, protocol-specific derived classes will be able to override its behavior, specifically the rule matching section.

Our base wrapper, thus, will be called with the following object properties:

- `standalone` defines whether the wrapper will act as a standalone daemon or will be called from the inetd daemon.

- `listenPort` holds the port number on which we will wait for an incoming connection. Of course, if it is not in standalone mode, this property will not be accepted, because when inetd calls a program to handle a request, the socket is already bound with `stdin/stdout`.
- `destType` indicates the type of connection to be opened to the server, whether it will be an IP connection or a pipe to another program residing on the same system.
- `destAddr` has the IP address to the host providing the real service. This can, of course, be the same host that is providing the wrapper. Care must be taken, however, not to allow outside hosts to get to the wrapped port. For an example on how to prevent this, please check the configuration presented in chapter 4. It will only be accepted if `destType` is set to IP.
- `destPort` holds the port number for the real service. This property, together with `destAddr`, defines the exact connection that will be initiated when an incoming connection arrives. It will only be accepted if `destType` is set to IP.
- `pipeCmd` indicates the command to pipe communication to . It will only be accepted if `destType` is set to pipe.
- `maxLineLength` is the most basic check that can be used for any given line

sent from the client to the server. This is a check that will be almost always done in order to prevent buffer overflows, so it is directly integrated in the base class. If maximum line length checking is not desired, this property should be set to zero.

- `logLevel` indicates the logging detail desired.

- `testLine` references the function that will test each line recieved from the client. If undefined, a default `testLine` function is provided by the base ProtoWrap, checking only for `maxLineLength`.

- `testReply` indicates whether the data recieved from the server should be checked or only the data generated at the client.

Keep in mind that, unless `testReply` is set to bidirectional checking, all the tests will be applied only to the data coming from the client to the server. The server will be allowed to send back to the client any data it wishes. The wrapper is not meant to be a traffic validator, only a helper for the server's security. `testReply` is provided for the cases where special states should be monitored at the wrapper and for the rare cases in which a response should be modified before getting to the client.

3.4. Interaction with the base ProtoWrap wrapper

It should be as easy as possible for the administrator to set up the wrapper for any given service or to modify its behavior. Thus, the commands issued should be kept to a bare minimum.

The first thing to include in a Perl program to be able to call the wrapper will be the proper `use` directive:

```
use ProtoWrap;
```

This will only call the base wrapper class (described in this chapter). To call a specific derived class, please refer to sections 3.5 and 3.6.

This way, we import all the wrapper's components into our program, and we are ready to fire a socket. Let us say we want to wrap an `imap2` connection - `imap2` is not implemented as a wrapper class in the sample classes included in chapter 4, but we want to protect the daemon from buffer overflows. The computer handling that particular service has assigned to it the IP address `192.168.0.1`, and will listen to the default `imap2` port, 143. We expect connections to be established very frequently, so the best strategy will be to run in standalone mode. We will limit lines to 100 characters, more than enough to receive imap2 commands. All

text sent over 100 characters will be silently dropped.

We now have all the information we need to create a new wrapper object. We do it by calling the `new` method for the wrapper:

```
my $imap2=ProtoWrap->new(
'standalone' => 1,
'destType' => 'ip',
'listenPort' => 143,
'destAddr' => '192.168.0.1',
'destPort' => 143,
'maxLineLength' => 100,
loglevel => 2
);
```

This way, the object has been created and is ready to start doing real work. Before starting it, it is advisable to check whether initialization was successful. If a needed parameter was omitted or an unexpected parameter was given, ProtoWrap will refuse to create an object, returning with `undef`[1] instead of an object. Thus, we need to do this test:

```
if (not defined $imap2) {
```

[1] The value `undef` is a special value used by Perl to indicate that a variable has no value. It is analogue to C's `NULL`. Note that `undef` is not equal to zero — `undef` is `undef`.

```
die('imap2 wrapper could not be started');
}
```

Now, the wrapper should be started. This is done with the startServer method:

```
$imap2->startServer() or warn 'Can\'t start wrapper';
```

That's all, we now have a working and listening wrapper listening to port 143 for a connection, ready to forward it to our server. The ProtoWrap process forks[2]. The parent process should remain as a running process after starting the wrapper in order to be able later to control it (i.e. kill it, modify its properties), so it should be put to wait. It is strongly recommended to use a passive waiting function, such as `sleep()`, because the parent will not be doing anything from now on. An exception handler can be added to wake up and do different things should a specific signal (such as `SIGHUP`, used to restart a daemon process) be recieved.

There should also be a way to stop a running wrapper, be it in order to reconfigure or restart it, or to temporarily suspend the service at the system administrator's request. We do this by issuing:

```
$imap2->stopServer;
```

[2] In Unix, a process can continue part of its execution in the background detaching itself from the main part by calling the `fork()` system call. A new process ID (`PID`) is given to the newly created process (referred to as the *child* process). The base process (referred to as the *parent* process) gets the child PID number and continues execution.

Although Perl handles garbage collection by itself[3], it is also useful and good practice to have a DESTROY method. Perl automatically calls the DESTROY method when the last reference to an object ceases to exist. Thus, when the object no longer serves its purpose, we can destroy it simply with :

```
$imap2 = undef;
```

There are other methods which can give information on or modify the wrapper's operation at runtime. They are:

```
%imapProperties = $imap2->getProp;
```

returns in the hash %imapProperties the actual values of all the object's attributes. This method is mainly for debugging purposes.

```
$imap2->set_maxLineLength($newValue);
```

changes the value of maxLineLength to the value contained in $newValue.

```
$imap2->set_logLevel($newValue)
```

changes the logLevel value to the value contained in $newValue.

[3]Garbage collection is the process of freeing up unused chunks of memory, closing files and sockets when they are no longer referenced and taking care of destroying unneeded objects. One of Perl's greatest strengths is that it will handle all the garbage collection for us. However, it is considered a good programming practice to always close what has been opened and destroy every unused object. This can also be useful when restarting or reconfiguring the wrapper: Instead of restarting the program or sending the necessary instructions to modify the current wrapper object's behavior, sometimes it is much easier to destroy the object and create it again.

```
$imap2->version()
```

returns ProtoWrap's version number.

3.5. Protocol-specific extensions: SMTP

The protocol that can best be served by this work is SMTP. In fact, this paper started originally being written with only SMTP in mind, but when the author realized other protocols could be vastly benefited with this work, he decided to also include them. SMTP, however, can be doubly benefited from the wrappers, both against DoS attacks and buffer overflows and, as we should soon discuss, spamming attacks.

3.5.1. The spamming problem

The early Internet did not suffer from spam. Spam did not appear suddenly, however, and thoroughly understanding its growth and behavior is the best way to put a limit to it.

Historical perspective

One of the most used —and thus, abused— services granted by the Internet is the electronic mail. Nowadays, anybody involved with technology has at least one electronic mail (E-Mail) address, and E-Mail is beginning to replace the role of the Postal Service in communicating people far apart.

E-Mail was first conceived more than 25 years ago, and since the mid-eighties its use has been widespread in the academic community. Several different mechanisms have been conceived to write, deliver and read the mail, but the most commonly used today is based on *SMTP* (Simple Mail Transfer Protocol), defined by *RFC* 822. This protocol is very lightweight, easy to understand and implement, and robust — but it is beginning to show its age.

In 1989, the commercial use of Internet was approved. Slowly but steadily, users all over the world —most of them with very little technical understanding— started using it. This has led to the Internet boom we are still living today — a highly successful arena with a place for every single topic that can be imagined. One of the activities which has flourished most is the electronic commerce. However, due to the large amount of online stores and businesses and the lack of high-quality sites, attracting customers has become a priority that must be fulfilled at any cost. The easiest and cheapest way for promotion is by E-Mail — An E-Mail message

needs to be sent only once, but it can be sent with hundreds or even thousands of destination addresses at a time. This is practically cost-free for the originating user, but it is not cost-free at all for the network operators and end users at all: The message is delivered to a server, which must process it and send it to all of the specified addresses, taking huge amounts of processing power and bandwidth. Hundreds or thousands of servers, when receiving the spam, must allocate their resources to it, storing useless and often large mails in their hard drives. Thousands of users paying an Internet connection have to download their mail —bloated by spam— to their personal computers.

These are the main reasons why server administrators all over the world are getting more and more concerned about spam.

Classification of spam

Spam is not one single problem. There are different kinds of spam, and each of them should be attacked in a different way. Here are the three largest divisions:

1. **Commercial spam** - Today, many businesses have flourished thanks to Internet, and the only place they exist in is the cyberspace. As any business, they want to get as much public into their "store" as possible. It was natural for them to start collecting E-Mail addresses and sending information about

their products. Slowly, they started collecting more and more addresses and building up huge E-Mail directories.

2. **Chain letters**, **jokes, virus warnings and other hoaxes** are another factor originating heavy loads of spam, sometimes called *benign spam*. The originators are the regular users: Someone finds something funny, interesting or something he feels as a threat (such as a new virus), and sends it to fifteen people. Each one of them forwards it to fifteen people, and they do the same. By the way, they usually send not only the original message, but append it with the address of **every** user that has so far received the mail, making the mail larger and larger, and unwillingly, revealing the addresses to possible commercial spammers.

3. **Mail bombs** - The widespread use of the *Microsoft Windows* operating system, together with their Office and Outlook programs, have contributed to create a new type of virus: a mail bomb. This is due to the programming language embedded in their Office applications, allowing *macros* —small programs embedded in a document— to be made part of otherwise text-only objects, such as a letter or a worksheet. Of course, macros can be any kind of code, including a malicious, self-replicating code: a *virus*. Macro viruses became widespread since 1996, soon after Microsoft began shipping their macro-enabled products. It was, however, not until April 1999 that self-

replicating E-Mail macro viruses appeared, the first one being called *Melissa*. Since then, this has become the favorite type of viruses written by even the most novice programmers — and, thanks to the addressbook capabilities of the popular mail user agent Microsoft Outlook, it has very quickly become one of the most widespread kinds of virus, because it automatically *jumps* to the mailboxes of all persons in the user's addressbook.

Conventional answers

A lot of effort has been put to stop spam — mostly unsuccessful. Sadly, spam is not a problem that can be stopped at a single site; every administrator of a SMTP-enabled server must do his best to stop spam, because it takes only one host that allows *relaying* to spam hundreds of other servers.

The first large-scale attempt to stop spam is found in version 8.8.8 of the popular *sendmail* SMTP server. With the standard sendmail distribution came a set of anti-relaying rules — most importantly, they were all turned on by default. Although this caused a lot of headaches for system administrators (who did not know whom to blame when their users could not send mail anymore from their desktop computers until they modified the configuration files for sendmail), it has helped significantly to slow down —although not stop— the problem.

Many sites decided to fine-tune their SMTP servers to disallow relaying, but many

did not do so. The problem remained the same as it was: a spammer knew he would not be able to send unwanted mail using a given server, but thousands of open servers remained all over the Internet. This led to a second move: The *RBL* — Real-time Blackhole List. This is a list of known sites that do not have anti-relaying rules installed, and since these sites are a favorite target for spammers, they are considered untrustworthy. Not trusting a site means discarding all the mail coming from it. This is not necessarily the best way out of the problem, because if the offending site has a large audience of users (such as the very well known case of the largest ISP in Mexico: *Telmex*), one will dump thousands of legitimate users' mail, only because they come from the wrong server. This is, however, one of the most popular ways of blocking unsolicited bulk mail nowadays.

Managing anti-spamming rules in sendmail, however, has proven difficult and intimidating for all but the most expert administrators. Sendmail is infamous for its difficult to understand configuration file, and rules are probably the most obscure part of this file. Using a wrapper, this can become an easy task — thanks to Perl's powerful pattern-matching capabilities, rules can be written in a format much easier to manage and understand.

3.5.2. Characteristics of a SMTP server

Before implementing a wrapper, a full audit of the base SMTP protocol was performed. The following are the conclusions found by the author:

RFC 821 and security

SMTP was first implemented when Internet was a network much different than the one we use today. Relaying back then was not frowned upon, as it was a legitimate and sometimes necessary way of sending messages to or from hosts that were not constantly connected. RFC 821 provides even specific commands for relaying, in the form

```
MAIL FROM:<>

RCPT TO:<@HOSTX.ARPA:JOE@HOSTW.ARPA>
```

This is the first point we should note — This syntax should not be allowed by the wrapper. Current Sendmail versions usually rejects this syntax, but it should not be overlooked.

Commands implemented

`MAIL`, `RCPT` and `DATA` are the three most important commands, indispensable in order to send a mail, so they must obviously be implemented. However, to implement them does not mean to blindly allow them - The greatest benefit in using wrappers is that we can monitor every single byte that goes into our system. We can watch for the following information in each of these commands:

- `MAIL` should check whether the user originating the mail comes from a RBL-blocked IP address and drop his connection; it can also check if the mail comes from an site authorized to relay. If it comes from an unauthorized site the wrapper should only raise a flag, for it may be a mail meant to be delivered to a local user.

- `RCPT` should allow setting a maximum number of message recipients - If we do not allow a message to be sent to more than ten people at a time we will make life harder for a potential spammer. Of course, some users may want to send a mail to many people at once, so we should not set this number too low. There must exist an option to ignore the number of recipients, in case an administrator prefers not to restrict this.

- `DATA` can limit a message's size. Additional checks could be made at this point, such as having a different behavior for different users or reacting to

a mail's content, not just to its envelope, but are not implemented in this version.

- `HELO` could check if the given address matches the originator's IP address and log it if it does not. However, most mail programs nowadays will still issue a HELO, but with a dummy address because they are often located on computers with no permanent IP address. It is a better decision just to check the format of the answer, and —of course— check it against maxLineLength.
- `RSET` can be cleanly passed on to the real SMTP server.
- `QUIT` can be cleanly passed on to the real SMTP server.

Commands omitted on purpose

Some commands' use should also be limited or completely prohibited. For starters, deprecated commands (`SEND`, `SOML`, `SAML`, `TURN`, `HELP`) should be blocked. Although they theoretically pose no danger, a basic rule in security is to be paranoid. If they are not to be used, they should be intercepted at the wrapper and never reach the real daemon.

`NOOP` is a similar case: This command does not need to get to the server. However, it can perfectly be implemented at the wrapper - we already know it will return a success (250) message.

VRFY and EXPN pose a different situation: These instructions should simply be blocked. The Internet is no longer a network we can trust in, and usually, when someone is querying for mail addresses it is an attempt to build a spamming directory, or to learn something about our system in order to try to crack his way in. The commands must not only block requests returning an error code indicating the address does not exist, but also log the attempt.

3.5.3. Implementation details

This wrapper is based on the base RFC 821definition of SMTP, and does not currently handle any extensions made to the protocol.

3.6. Protocol-specific extensions: POP3

POP3 poses much a simpler problem than SMTP. POP3 is a very simpler protocol, and only basic security checks should be performed to it, thus keeping the size of the wrapper much smaller than the one used for SMTP.

3.6.1. Characteristics of a POP3 server

Before implementing a wrapper, a full audit of the base POP3 protocol was performed. The following are the conclusions found by the author:

RFC 1081 and security

POP3's weakest point from a security standpoint is that the password is transmitted in clear text, allowing any potential attacker to sniff the network for passwords. This limitation, however, can not be handled by a wrapper, as the wrapper must still adhere strictly to the POP3 protocol to be useful; doing otherwise would make it necessary to write new POP3 clients, or perhaps even redefining the protocol. A couple of points in the AUTHORIZATION state can be strengthened by the wrapper:

The RFC specifies that after the `USER` command is specified the system must check if the user exists, and return `+OK` if a valid username was given, or `-ERR` if the specified username does not exist. This can unknowingly disclose information about the server's accounts. Quoting from RFC 1081 [7]:

> USER name
>
> Arguments: a server specific user-id (required)
>
> Restrictions: may only be given in the AUTHORIZATION state after

the POP3 greeting or after an unsuccessful `USER` or `PASS` command

Possible Responses:

```
+OK name is welcome here

-ERR never heard of name

Examples:

C: USER mrose

S: +OK mrose is a real hoopy frood

...

C: USER frated

S: -ERR sorry, frated doesn't get his mail here
```

Most POP3 servers nowadays choose to always give a positive answer to `USER`. Some, however, will still answer negatively. The wrapper should intercept this reply, giving always a possitive answer for this command.

A POP3 server does not need to put a limit to unsuccessful login attempts, according to the RFC. This can be used by an attacker to try to use brute force to guess somebody's password. The wrapper will take two steps to prevent this: First of all, it will limit to a user-set number the number of failed login requests that can be allowed in a single session. Of course, a new session can be established, but it is an extra hurdle for the attacker, an automatic program will be somewhat harder to do. The wrapper will also add some waiting time every time a password is not

correctly provided —two or three seconds is advised—, making it much harder to perform a large number of login attempts.

When the connection progresses to the TRANSACTION state there are still some points that should be monitored. First of all, the number of valid messages. Whenever a `STAT` is performed byt he user, ProtoWrap will not only report the answer to the client, but will also store the result. This is to learn the number of messages waiting to be retrieved. If the client tries to access (via `TOP` or via `RETR`) a message below the first one or after the last one, or does not specify a valid number. Each time the `DELE` command is used, the deleted message number should be added to a hash, to prevent the deleted messages to be requested again. Of course, if a request is denied by the wrap it should never reach the server.

Commands implemented

Most commands will be allowed to pass through with no modifications, checking just the number and format of the arguments. The checks performed will be:

- `USER` will only be allowed in the AUTHORIZATION state. Will be passed if one and only one argument is specified. If the last command specified was `USER` (no password was specified), or if the maximum number of login attempts has been exceeded, it will be denied.

- `PASS` will only be allowed in the authorization state, after a `USER` command has been issued. Will be passed if one and only one argument is specified. Each time it is called it will check and increment the counter for maximum login attempts.

- `QUIT` will be passed with no arguments. If arguments were specified by the client, they will be chopped.

- `RSET`, `LAST`, `STAT` and `LIST` will only be allowed in TRANSACTION state. They will be passed with no arguments. If arguments were specified by the client, they will be chopped. `RSET` will clear the deleted messages table.

- `RETR` and `DELE` will only be allowed in TRANSACTION state. They will only be allowed with one and only one numeric argument. This argument will first be checked for validity against the list of deleted messages, disallowing already deleted messages, and the maximum message number.

- `TOP` will only be allowed in TRANSACTION state. It will be allowed with two and only two numeric algorithms, and only after checking the first argument against the deleted messages table.

Commands omitted on purpose

- `RPOP` will be denied, as it is based on the presence of the `.rhosts` and `hosts.equiv` files, an obsolete and insecure authentication method which often leads to security compromises.

- `NOOP` can safely be implemented in the wrapper and never reach the server. The wrapper will send a positive answer to the client.

3.6.2. Implementation details

This wrapper is based on the base RFC 1081 definition of POP3, and does not currently handle any extensions made to the protocol.

3.7. Protocols deliberately not contemplated

ProtoWrap is a quite comprehensive and ambitious project. However, there are many kinds of protocols not well suited to be wrapped by it. The most representative examples are given here.

3.7.1. HTTP

HTTP is also a TCP-based line oriented protocol, so it could easily be wrapped with ProtoWrap. It is, however, very different from the other protocols analyzed. Both SMTP and POP3 have a session during which each command given by the client will be answered by the server, which will then wait for further input from the client. HTTP connections usually recieve information for a short period in the beginning of the connection, and send the result of the request (which can be even several megabytes long) afterwards. The data sent back to the client can sometimes be handled by a line-oriented wrapper (for example, when sending a text or HTML file), but the bulk of the content will be binary data - mainly images. Wrapping it would heavily increase the amount of RAM needed, and lead to probable DoS attacks.

3.7.2. FTP

The File Transfer protocol, defined by RFC 114, and updated by RFCs 141, 171, 172, 265 and others, implements a way of communicating data that makes it very hard to be wrapped: Not only one port is opened. The client connects with the server, and the server connects using a second channel back to the client. This allows for separate data and control channels - and makes it much harder for the

wrapper to function. If the wrapper opens a connection in behalf of the client to the server, then the server will try to connect back to the wrapper's IP address for the data channel. This channel would not be line-oriented, so it would have to be dealt with using a different strategy.

The author reccomends to all administrators wishing to protect their FTP daemons against DoS and buffer overflow attacks to set up the ProFTPD package, available for different Unix platforms, as it currently is the most secure freely available FTP server.

3.7.3. Telnet

Although not a complete protocol *per sè*, Telnet is a very popular way of communicating. It does have great shortcomings —specifically, the fact that all the information is sent in clear text between hosts— but they can not be addressed by this wrapper. Most notably, Telnet is character-oriented, while this wrapper is line-oriented.

If encryption of telnet sessions is required and using SSH is for some reason unacceptable, I suggest considering SSLWrapper[9].

3.7.4. SSH

Even if the communication to be transmitted were to be line-oriented, the very essence of an encryption algorithm would make it impossible for us to wrap it. Not only must the content be indistinguishable from garbage, it is —as Telnet— character oriented. Having a wrapper able to detect attacks against this server would be very important, as buffer overflow attacks against SSH linked against the RSA library have been found and not yet been fixed, but it would require a completely different approach than the one this program has, and would require being specific to SSH, unable to be easily extended to support different protocols.

3.7.5. UDP and ICMP based protocols

UDP

While with all the previously treated protocols we can rely on the session management the TCP layer provides us, there are many protocols that will not give us such a facility —Protocols based on UDP, such as NFS, ICQ and all RPC portmapper-controlled protocols. UDP (*User Datagram Protocol*) was designed to work for services that need to be resistant to changes in the network topology, have fast recovery times, and do not need explicit session management.

Working with UDP would force us to define timeouts and open even more our system to DoS attacks, which would be extremely simple to carry out: Sending a large number of requests would perfectly do it. Although this can be done, the suggested implementation would perform poorly.

ICMP

ICMP (*Internet Control Message Protocol*)is a protocol meant to be used as an aid for other protocols, as well as system administrators, to test for connectivity and search for configuration errors in a network. ICMP requests must be quickly answered in order to maintain confidence in their results. Systems also continuously receive ICMP requests, so wrapping them would prove extremely heavy for our resources. In ICMP requests, the information contained in the packet is seldom relevant, so not only processing its information becomes too hard, but it also proves completely useless. If ICMP filters are to be set up for a system, they should do so only at the source/destination level, handling it directly from the kernel.

3.8. SSL encrypted protocols

SSL (*Secure Socket Layer*) holds an intermediate position between SSH and traditional protocols: It adds an extra layer to the connection, encrypting the com-

munication between the client and the server, but not requiring the protocols used to support encryption. SSL can be applied to any TCP-based protocol, and it can effectively be handed over to the wrapper, which would filter it and hand it to the real server. However, if the wrapper and the server were not to be located on the same computer, this would completely defeat the encryption done by the SSL, sending the information in clear text over the network to the server.

For further information on SSL, a good place to start is the OpenSSL project's web page[8], as well as the SSLWrapper project's web page[9]

3.9. Validity of data

The facts presented in this study have been gathered from many security sources, including personal experiences of various systems administrators and programmers, different security mailing lists, conferences, magazines and books.

The solution suggested in this work, although formally justified and having been presented at conferences and discussed with peers, has *not* been thoroughly tested, though, so once again it is necessary to insist that it should still be considered a proof of concept. Different implementations of this idea can be more efficient, more secure or better programmed overall. However, the author is confident the actual implementation is good enough as a base for future work, and, with not

too much extra work, can become a real, useful security tool.

3.10. Originality and limitations of data

It is very important to be aware of the details on originality and limitations before using or working on ProtoWrap.

3.10.1. Originality

ProtoWrap is a completely original and independent project, developed by Gunnar Wolf. There are some programs already adressing some of the same issues Protowrap is aimed to, such as Wietse Venema's *TCP Wrappers* [3] or Marcus Ranum's *nmap* [4], and their success was motivator to the development of ProtoWrap. Their approach, however, is quite different from Protowrap, and Protowrap can be considered a completely original product.

Protowrap has been presented and discussed in several forums, including UNAM's international conference *Seguridad en Cómputo 2000* and diferent mailing lists.

3.10.2. Limitations

ProtoWrap is highly vulnerable to suffering DoS attacks, given the amount of processing that can be dedicated to each line. Each systems administrator must weigh what is more important to his site's operation —High availability or high security. Most systems administrators will choose security.

As ProtoWrap is built centered on security and RFC adherence and not on supporting extensions to the protocols, the extensions will be treated as erroneous commands and discarded. This might make some client-server interactions slower and some operations impossible, but —once again— this is a compromise that many system administrators will be willing to take.

In its current status, ProtoWrap is not easy to set up. Anyone interested in running ProtoWrap must craft their own startup script. In order to allow systems administrators without Perl knowledge to set it up, a startup script reading a configuration file is among the top priorities for future development.

Due to its licensing, the code to ProtoWrap can not be included in propietary products — If ProtoWrap is enhanced or included in another product, the changes or the product will be made available under the GPL license.

3.11. Summary

Chapter 3 introduces the implementation and usage of ProtoWrap. The base ProtoWrap class is object-oriented. New wrapper objects are created by specifying the `standalone`, `destType`, `listenPort`, `destAddr`, `destPort`, `pipeCmd`, `maxLineLength`, `logLevel`, `testLine` and `testReply` attributes — some of them are required, some are optional, please refer to section 3.3 for more details. Once the object is created, the wrapper can be started by calling the `startServer` method of the newly created object:

```
$wrapper->startServer() or warn 'Can\'t start wrapper';
```

The method `stopServer` should be used in the same way to stop the server without destroying the object. In case the object needs to be destroyed, assigning any value (being `undef` the most obvious value, to avoid confusions) to it will stop and destroy it.

For more details on the methods included in the basic ProtoWrap class, see section 3.4.

Two protocol-specific extensions are provided as part of this work, to wrap SMTP (details in section 3.5) and POP3 (details in section 3.6).

Section 3.5 starts bye giving an introduction on the problems SMTP suffers, specifically spam (subsection 3.5.1). In subsections 3.5.2 and 3.6the author analyzes the

possible vulnerabilities implicit in a server fully complying to RFCs 821 and 1081, and explains the criteria followed while programming the ProtoWrap extensions.

There are many protocols which are not suitable for this kind of wrapper, and they are analyzed in section 3.7. Most of these protocols are discarded because they are not line-oriented (telnet, SSH), work in such a way this wrapper would work suboptimally with them or would not work at all (HTTP, FTP), or are not transported over TCP (UDP, ICMP). Section 3.8 gives a quick overview of how this wrapper should —and should not— work together with SSL encryption.

Sections 3.1, 3.2, 3.9 and 3.10 give a formal framework for this work, detailing the approach, process and observations on the results.

4. Implementation

This chapter talks about design and technical issues found by the author while developing the ProtoWrap code. The following observations should be very important when deciding towards or against using ProtoWrap in a production environment, or when planning to write a new ProtoWrap module.

4.1. The dual I/O problem

Among the first problems that surfaced while coding ProtoWrap was the dual I/O problem: Data can come, at any time, from either the client or the server. Many different models were tested, being the most important ones:

Forking+IPC As soon as two connections are established, the wrapper forks into

two separate processes; the parent handles communication with either socket and the child with the other one. Now, this would work if the amount of validation were to be limited. Handling, however, more complex client/server interaction requires mantaining a state of the connection, which can only be achieved by having constant communication between the processes. This could be achieved fairly easily using threads, but as thread support in Perl is still regarded as experimental, the author decided to go for a more reliable solution. IPC signals, temporary files or other external methods are prone to becoming too slow, and would make the code far more complicated.

Preset whole lines The wrapper knows beforehand how many lines does each command answer with, or how to determine when the last line is reached from the data stream itself. This could possibly be implemented without much hassle for different protocols, but a simple programming error can lead to problems much more dramatic and harder to spot. Depending on the protocol involved, a lockup state could easily be reached (for example, recieving even a single character from the client, without recieving an end of line afterwards, will never allow the server to talk again), making our server keep possibly hundreds of open, idle TCP connections, and maybe even locking other systems depending on our correct behavior.

Low-level The wrapper could also talk with server and client not using the easy-

> to-understand, high-level socket interfaces (using Perl's `IO::Socket` module), but instead treating them with the lower level filehandle interface (with Perl's `IO::Handle`). By doing this, we can use `IO::Select` to handle I/O from multiple filehandles in quite an elegant way.

The low level model was chosen because of its high effectivity and low intrusiveness.

4.2. Dealing with different data sources

The first step was to get ProtoWrap to work with one model of communication. The first path taken was using socket communication on both ends: The wrapper became a TCP daemon at startup, and when a connection is recieved, a new TCP connection is created to the server. Although this will be the most common setup, it is very important to provide at least two other basic features: The ability to be started from inetd (in order to be able to bind to low ports without requiring root privileges, avoiding also being always loaded in RAM for seldom-used services) and the ability to invoke the server locally, handling I/O via the STDIN/STDOUT filehandles (allowing us not to open a TCP connection, and effectlively avoiding any external user from getting to the actual server).

Achieving this was a challenge: In order not to duplicate the code that handles communication, we will handle two filehandles per connection, one to be used for

reading ($client_rd and $server_rd) and one for writing ($client_wr and $server_wr). If we are working with sockets (a real TCP connection), the read and write filehandles will point to the same actual object, but they need to be different when dealing with any other connection type.

Process spawining behavior must also adapt to different invocation styles: When the wrapper is invoked standalone, as soon as it is started it forks to detach itself from the main program. This is definitively not wanted when called from inetd. Programs called from inetd do not close the connection by closing a socket —they close the connection when their execution finishes. If the program forked, it would never close the connection, and the client could be left hanging forever. Thus, the startServer function appears as follows:

```
sub startServer {
    my $self = shift;
    my ($error,$child);
    $error=0;
    if ($self->{standalone} == 1) {
        $self->{listSock} = IO::Socket::INET->new(
            LocalPort => $self->{listenPort},
            Type      => SOCK_STREAM,
            Reuse     => 1,
```

```
            Listen    => 10)
            or $self->log(0,
            "Couldn't be a TCP server on port $self->{listenPort}") &&
               ($error = 1);
        return undef if $error;
        $self->log(0,"Can't fork to accept incoming connections!: $@ $!")
               if (!defined($child=fork()));
        if ($child == 0) {
            # Child process - take care of the connection
            $self->getConn();
        } else {
            # Parent process - return the child process' PID
            $self->{pid} = $child;
        }
    } else {
        $self->getConn();
    }
}
```

In the main program loop a `fork()` must also be avoided when called from inetd. The wrapper will not wait for an incoming connection — The connection is already

there when the program is invoked. If a `fork()` happened as it usually would, the server would be called over and over again, exhausting the computer's resources. To avoid this, we fool the program into thinking it has already been forked and the child process is now being executed:

```
if ($self->{standalone} == 1) {
    # Standalone mode - Every recieved connection should get
    # its own server
    $self->log(0,"Can't fork to handle incoming client connection!:
        $@ $!") if (!defined ($child=fork()));
} else {
    # inetd mode - The forking is done by inetd, so we act as
    # the child process
    $child = 0;
}
```

4.3. Reimplementing server functionality

When wrapping a service, unfortunately, some of the functionality already provided by the server must be reimplemented. Fortunately, in most cases, it is possible not

just to reimplement it, but to enhance it. This particular experience springs from the SMTP wrapper.

4.3.1. Why must functionality be reimplemented

Recent versions of most popular mail transport agents (*MTA*) started including anti-relay filters to help fight spam turned on by default. Most system administrators all over the world decided to allow relaying to their internal networks' computers, while denying it to every external computer.

When using a wrapper, every connection appears as if it came from the host where it is located. Of course, this feature of the MTAs is no longer useful, and it should be reimplemented at the wrapper. Fortunately, as everything is already arranged by stages (see section 3.5.2 for explanation, section A.3 for the code), adding this proved extremely easy.

The world's most popular MTA is *sendmail*, which is, at the time of this writing, at its revision number 8.11.1, handles access lists similar to the following example:

```
localhost.localdomain          RELAY
localhost                      RELAY
127.0.0.1                      RELAY
mynetwork.net                  RELAY
```

```
192.168.1                 RELAY
172.16.50.30              REJECT
10.3                      REJECT
spammer@hotmail.com       REJECT
advertisement.net         REJECT
```

This file is usually located at /etc/mail/access, and must be compiled into a DBM hash called access.db in the same directory to be used. The file is Tab-separated; the first column holds the patterns and the second one holds the action associated to it.

In this example, relaying is allowed to the localhost (lines 1, 2 and 3) and to the local network — every host belonging to the mynetwork.net domain (line 4) or with an IP address belonging to the 192.168.1 network (192.168.1.0 through 192.168.1.255, line 5).

On the other hand, input coming from the specific host 172.16.50.30 (line 6), or from any host belonging to the 10.3 network (10.3.0.0 through 10.3.255.255, line 7), to the specific mail address spammer@hotmail.com (line 8) and to the whole advertisement.net domain (line 9).

4.3.2. Matching the server's lost functionality

Perl can do a much better job than this file: When starting ProtoWrap for SMTP, it will accept the attributes `relayIpList`, `relayDomainList`, `blockIpList` and `blockAddrList` as as references to lists . As part the mission for ProtoWrap to exist was to stop spam, the author decided to follow a traditional approach in firewalls: Everything that is not expressly permited is forbidden. Thus, ProtoWrap will accept only what is specified in this file. A basic equivalent configuration to the above example, starting a wrapper to a local MTA and listening to port 20025, would be:

```
$wrap = ProtoWrap::SMTP->new(
    'listenPort' => 20025,
    'destAddr' => '127.0.0.1',
    'blockAddrList' => ['spammer@hotmail.com',
        'advertisement.net'],
    'blockIpList' => ['172.16.50.30','10.3'],
    'blockBodyList' => ['^Subject: .*ILOVEYOU'],
    'relayIpList' => ['127.0.0.1','192.168.1'],
    'relayDomainList' => ['localhost.localdomain',
        'localhost','mynetwork.net']
```

```
);
```

4.3.3. Enhancing functionality

Although more compact and —at least to some users— easier to understand, this example simple still does not show the power of Perl. Using a sendmail configuration file, if an administrator noticed spam coming from the domains spammer.org, spammer.net and spam4you.com, a single rule would match them all, thanks to regular expressions: `'spam.+'`. However, this would also block all legitimate mail from spamstoppers.org — this can be solved either by excluding specifically this domain with `'spam[^(stoppers)]'`, or by including only the three offending domains, with `'spam(4you\.org|mer\.(org|net))'`. Regular expressions can become, as seen here, a bit difficult to read for the new user, but they are a great, powerful tool for the seasoned administrator.

ProtoWrap not only handles the regular expressions in order to reimplement Sendmail's functionality lost while wrapping, it also implements something else, not contemplated by Sendmail: Content inspection.

Mail-based viruses have become commonplace. They are usually very fast to spread and to damage information. They do this, however, with patterns that are very easy to find. The infamous 'I Love you' virus will be taken as an example. This

virus spreads as a `.VBS` (Visual Basic Script) attachment to email messages, all of which have 'ILOVEYOU' as a subject. Sendmail can stop this virus with the following lines located in the `sendmail.cf` file:

```
HSubject: $>Check_Subject
D{WORMmsg}Access Denied - This message may contain a virus.


SCheck_Subject
RILOVEYOU                $#error $: 501 ${WORMmsg}
RRe: ILOVEYOU            $#error $: 501 ${WORMmsg}
RFW: ILOVEYOU            $#error $: 501 ${WORMmsg}
RRead: ILOVEYOU                  $#discard
RNot read: ILOVEYOU              $#discard
RDelivered: ILOVEYOU             $#discard
RUndeliverable: ILOVEYOU         $#discard
```

The steps followed are:

1. H — Search for something in the header. In this case, search for the string Subject:, and call this header rule Check_Subject
2. D — Define an answer, called WORMmsg, and the text sent to the requesting client when it is invoked.

3. S — Define a search. From here on, we will be talking about the line defined in step 1.

4. R — React to the string found by sending an error (first three R lines) or by discarding the message (last 3 R lines)

Note that no pattern matching is actually done; this is done only by exact-matching.

The same effect can be simply achieved using ProtoWrap, by adding the string `'^Subject:  .*ILOVEYOU'` to the `blockBodyList` attribute. This one will, of course, catch every possible modification done by a client program (for example, prepending `re:`, `Re:`, `RE:`, `ref:`, `Ref:`, `REF:`, `fw:`, `Fw:`, `FW:`, etc.) — a mounstrous task if using straight Sendmail syntax.

4.4. Excessive logging and privacy concerns

ProtoWrap can log its activity in five different verbosity levels. This was done in order to allow the system administrator to have a proven wrapper be as quiet as possible, reporting only fatal errors and exceptions, or to be helpful when developing new modules or debugging incorrect operation, reporting every single byte

and the way it was reacted upon. All the logging is done using the central Unix Syslog facility.

Being too verbose raises some privacy concerns - *everything* gets recorded. User passwords, the whole text to users' email messages, everything. It is strongly suggested *not* to use the three more verbose levels (3, 4 and 5) except when absolutely necessary.

Besides the privacy concerns, the logs get saved to a hard disk. Super-verbose logging has a lot of overhead, giving very abundant extra information on each line. Logging too much can easily fill up the logging partition, and valuable log data could be lost. An attacker could easily abuse this to erase the tracks of what he had just done to the system.

Reading logs is also a very important daily task that every administrator should religiously do. If the logs are too large, important information can be missed, endangering the whole system.

On the other side, being too conservative with the amount logged can lead us to ignore useful information.

The author strongly reccomends a log level of 1 or 2 for normal operation, and 3 or —at most— 4 for normal debugging. Level 5 (ultra-verbose) should only be used to monitor behavior while developing new modules. Level 0 should be avoided, as

it leaves very important information not logged..

4.5. Attaining low ports with low privilege

It is very useful —sometimes outright necessary— to start the wrapper with high privileges. This can be the case when running in standalone mode for services which listen to ports below 1024, which in Unix require root access to be opened, or when running in pipe mode, and the server program requires running as root —This should not be the behavior when running the wrapper! Were an attacker able to subvert the wrapper, it is one of the author's design goals to give him as little access as possible.

The approach followed is to allow the user to specify which UID the wrapper should run as - and to deny any attempts to run with root.

There are two main point at which the program should be able to drop privileges, and the decision on when to do so depends on each specific case. Therefore, the following attributes were added to the base ProtoWrap class: (definitions from the code's internal documentation)

`setUidTo` Which user ID should the wrapper run as (numeric). Defaults to current UID. Will not allow 0 (root).

`runSrvSuid` Whether to run the server program as root (1) or to drop privileges just after acquired the listening port (0). Defaults to 0. Useful only if running with `standalone==1` and `destType=='pipe'`, ignored otherwise.

The user specifies in `setUidTo` which UID to switch to, and in `runSrvSuid` when to do so. If `runSrvSuid` is set to 1, the server program will have to be run as root, so the privileges should be preserved until the server program has been started, otherwise privileges will be dropped just after the privileged (low) port is open and ready to accept connections, and before any forking takes place.

Of course, if `destType` is `'ip'` or if `standalone` is 0, specifying when to switch the UID does not make sense anymore - If `destType` is `'ip'`, the wrapper is not starting up the server, so it cannot affect what user will it run as. If `standalone` is 0, no listening sockets will be opened, and the only point at which privileges could be dropped is when starting up the server process. If both conditions are met, there is no point in starting the wrapper with the root UID, as it will not perform any privileged operations.

4.6. Auto-looping

Consider the following case:

```
#!/usr/bin/perl
use ProtoWrap;

my $wrap = ProtoWrap->new('standalone' => 1,
  'listenPort' => 3000,
  'destType'   => 'ip',
  'destAddr'   => '127.0.0.1',
  'destPort'   => 3000
  );
die 'Can\'t start SMTP wrapper' if (not defined $wrap);
$wrap->startServer() or warn "Can't start wrapper";

exit 0;
```

We have here a wrapper that will connect to itself. Any user could launch it, because it binds to a high port. Nothing unusual would happen right away, but as soon as the first packet for that port arrives, a cascade of `fork()` system calls will fall down on the system, effectively making it unusable after just a couple of seconds.

The author, however, decided not to prevent users from doing this — it would require huge amounts of code, slowing down normal execution, and it could always

be circumvented by altering little bits of this sample invocation. Users to local access to a computer can very easily create DoS conditions without resorting to ProtoWrap — Consider any of the following examples:

- `while (1) { fork(); }` would very quickly fill up the process table, forcing the administrator to shut down the computer, or take very tedious steps to kill hundreds of useless processes.
- `$a = ' '; while (1) { $a .= $a; }` fills up all the available memory —physical and swap— and must, once again, be killed by the administrator, if he can get enough memory to even get a shell!
- `while (1) {}` does not make the computer not usable - and maybe this makes it the most dangerous of the examples here presented. This idle loop is very processor-intensive, and slows down the computer significantly. However, the system continues to operate, and most administrators will consider this as usual, correct behavior! Although some Unix schedulers will notice the process is a CPU hog and will lower its priority in the ready queue, it will continue slowing down the system, consuming very important resources.

They are all much simpler to implement than the first one, much more effective and harder to notice.

5. Conclusions

In order to test ProtoWrap and measure its effectiveness, the author set up some wrappers for the servers at his charge. The resulting observations are also to be found in this chapter.

5.1. Future enhancements to ProtoWrap

As explained before, this work should be considered a proof of concept, not a finished work. The following ideas are to be implemented in the future, to make ProtoWrap become more a truly useful and manageable tool than an abstract idea.

5.1.1. Master initialization program and configuration file

Although using ProtoWrap is not hard, the user must know some Perl programming in order to configure it. The author's goal is to improve ProtoWrap until it reaches a usability level similar to the `inetd` daemon: Having a simple to understand configuration for wrappers in standalone mode, in a format such as the following:

```
79 pipe /usr/sbin/in.fingerd
143 ip 192.168.10.34:50143
```

Of course, this would have to be adapted in order to handle protocol-specific extensions, such as the ones provided for SMTP or POP3. Maybe a better syntax would follow the one introduced by xinetd, which uses blocks instead of lines for each service's configuration. A configuration similar to the one proposed below would become:

```
finger {
listenPort = 79
destType = pipe
pipeCmd = /usr/sbin/in.fingerd
}
imap {
```

```
listenPort = 143
destType = ip
destAddr = 192.168.10.34
destPort = 50143
}
```

Configuration files would, of course, become longer. A quick overview of the configuration would be harder to achieve, but the format allows for a much more flexible and extensible syntax, able to grow as new extensions are developed or new capabilities are added to the base class.

Any chosen way will make ProtoWrap much more attractive and easy to use for novice systems administrators, and reduce the errors in which even seasoned administrators would fall if the configuration had to be coded in Perl.

5.1.2. Signal handling

Unix processes often interoperate by using *signals*. In a Unix system[1], there are many predefined signals, and probably the three that are more frequently used are `SIGTERM`, `SIGKILL`, `SIGHUP` and `SIGCHLD`. Signals provide a way of controlling

[1] For a complete list of signals, refer to the signal(7) man page on most Unix and Unix-like systems. 19 signals are defined in the POSIX .1 definition, and many more are supported by most systems.

processes, either from other processes or from the kernel itself. Some signals can be ignored or *caught*[2] and some can not. The signals just mentioned have the following purposes:

SIGTERM Termination signal. A process which recieves this signal *should* finish its activity and shut down cleanly. This signal can be caught or ignored.

SIGKILL Kill signal. When this signal is sent to a process, the process gets immediately terminated. This signal can neither be caught nor ignored, and it is often used as the last resort method to kill a process.

SIGHUP Hangup signal. This signal can be used in many different ways; it was originally concieved to inform a process that its controlling terminal or process has just hung up or died, and that it should terminate cleanly. Another use has however been found for it due to its ability to be caught, and has now become its primary use: Sending SIGHUP to a daemon process makes it reload its configuration and restart.

SIGCHLD Child stopped or terminated. This signal is sent whenever a child process has finished execution, so that the parent process can clean up after

[2] Catching a signal means that when the signal arrives to the process, the process can decide to call a function to handle the event instead of following the default action, which usually is to terminate the process

it, recieve the results to a certain operation, or whatever information must be sent. This signal is often ignored.

Currently, ProtoWrap does no signal handling, except for specifically ignoring SIGCHLD to avoid having zombie processes lying around waiting to be *reaped*. Restarting on SIGHUP can be very useful, and it should soon be added to ProtoWrap. As ProtoWrap does not hold long-term information on system memory, catching SIGTERM would not be really important, except for logging or cleanup purposes.

5.1.3. Load balancing

Another possibility that opens up when designing a program like ProtoWrap is to implement in it a load balancer. Load balancing is a technique very often used in heavy-traffic sites, allowing many computers to answer at the requests directed at a single IP address/port pair. There are many ways to achieve this:

Round-robin DNS The DNS server is configured to relate more than one IP address to each hostname, as in the following case:

```
[user@host user]$ nslookup www.microsoft.com
Server:  dns1.unam.mx
```

```
Address:  132.248.204.1

Non-authoritative answer:
Name:    www.microsoft.akadns.net
Addresses:  207.46.230.229, 207.46.230.219, 207.46.131.199
          207.46.230.218, 207.46.130.45
Aliases:  www.microsoft.com
```

Clustered server There are many ways to build a cluster[3], suited for different purposes. Among the most often heard is *Beowulf*, a project started at NASA which implements *Parallel Virtual Machines*. However, clustering servers do not require (in fact, would hardly be benefited from) a programming technique as hard to master and code; a *fork-and-forget* clustering system such as Mosix[4] is much more adequate — In such a system, a master server appears to be the only one running, but when a connection is recieved, the server process is forked to a different computer, according to the systems' current load. This may be the most scalable solution, and it provides the closest answer to high-availability requirements.

[3] A cluster is any number of computers which appear to be only one, either by using special hardware architecture or software dedicated to join them.

[4] for more details, visit http://www.beowulf.org/ for Beowulf, http://www.mosix.cs.huji.ac.il/ for Mosix

Dispatching server This will be the proposed solution for extending ProtoWrap:

> A single server is registered, and no special software is running in each server. A separate machine appears to be serving all the pages, but what it does essentialy is to wrap the connections and forward them to another server.

ProtoWrap can easily be expanded to become a load balancing dispatching server — most of what is needed is already done. Of course, as a great benefit, it would be also a traffic validator, ProtoWrap's main function, protecting all the real servers.

How would this implementation look like? ProtoWrap's code could be easily modified to discern whether the value contained in the `destAddr` attribute is a scalar value (as it usually is) or a reference to an array of values. When creating the wrapper, instead of

```
destAddr => '192.168.0.1'
```

we would have

```
destAddr => ['192.168.0.1', '192.168.0.2', '192.168.0.3']
```

Load balancing extra features

Of course, when implementing a load balancer, care must be taken to implement a *good* load balancer. Although it is not ProtoWrap's main goal nor focus, some

points should be taken into account:

- Load balancing is closely related to high availability. If a system in our sample pseudo-cluster were to fail, it would be disastrous to have one third of the requests be forwarded to a dead server. Not only would it give a bad image for our service, but it would also hog down ProtoWrap's server, having it handle a large number of TCP connections waiting to be established. A possible mechanism would be to have the parent process ping the server machines at frequent, regular intervals, and as soon as one of them does not answer the ping, remove it from the destAddr list. Of course, once it is back online, add it back to the list.

- A good load balancer does not only send a packet to each server. A good load balancer keeps track of how each of the servers is responding and sends each following request to the least loaded one — at least, allows the administrator to specify the priority for each computer (for example, a 800MHz Athlon should recieve at least twice the hits as a 500MHz Pentium II). An extension could be written to periodically query the servers for their load average, or the simpler approach (allowing the administrator to specify priorities when creating the wrapper) could be used. Maybe allowing the administrator to specify the same IP address more than once would be enough for the most basic needs.

Anyway, load balancing can become a very useful characteristic for ProtoWrap, and it will very likely be added in the near future.

5.1.4. Perl-style (POD) documentation

ProtoWrap has been designed to be used as a Perl module. Although the author has done his best to have the code thoroughly commented, and commented code is often regarded as the best possible form of documentation among programmers, Perl modules implement a very useful and easy documentation format called POD (Plain Old Documentation), which is written in the same file as the code, and when the module gets installed in the system using the standard Makefile.PL A.1 method gets translated into a standard man page.

Perl programmers often take POD documentation as a starting point to begin using a newly installed module. Thus, most of ProtoWrap modules' blocks of comments should be converted to POD. This step was not yet taken because, as it has been repeated throughout this text, ProtoWrap is presently a proof of concept, and emphasis has been put on making the comments illustrate as clearly as possible the inner working of the module, in order to have as much peer review as possible. The POD code format is not as natural to programmers as regular comments are.

5.2. Sample configurations

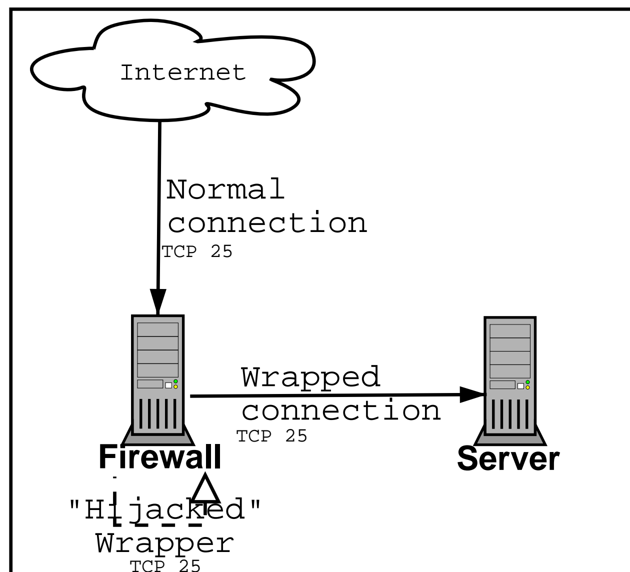
ProtoWrap has been coded in a flexible enough way to allow for very different ways to solve the problems described in chapter 1. The most common ways of calling ProtoWrap will be described in this section. They will be accompanied by sample code showing how such a scheme would work. For the sample code, we will assume that the server's address is 192.168.0.1, listening both to port 25 (SMTP) and 110 (POP3). Servers will be running Linux distributions based on kernel 2.2.*x*. If a firewall is assumed, it will be set up with OpenBSD 2.8 (`ipf/ipnat`), and its IP address will be 192.168.0.254[5]. The LAN will be connected on interface ep1, and the Internet connection will be on ep0. The port 10025 was arbitrarily chosen as the port on which the wrapper will be listening on.

Of course, a single system can run with many of the following configurations at the same time, with each service having a different behavior.

[5]The IP addresses were not randomly chosen; networks 10.x.x.x, 172.16.x.x to 172.31.x.x and 192.168.x.x are reserved for private, internal use. For more information on this, please check RFC 1918, *Address Allocation for Private Internets*, Y. Rekhter et. al.
There is also an implicit standard between many network administrators, making their routers and firewalls use the highest available IP addresses of each subnetwork.

5.2.1. Wrapper running at the firewall

A firewall, located at a network's perimeter, either between the network and the Internet or between the servers' segment and the rest of the network, can very easily control many of the servers' network ports.



Such a scheme would result in the following advantages and disadvantages:

Advantages

- A single, central configuration

- Upgrades can be done only once, they will be automatically applied to every server inside the firewall
- All logging is centralized on a single machine, so detecting attacks to multiple servers becomes easier
- Firewall becomes a transparent proxy
- The servers' configuration must not be modified, the server programs will run as usual. All the wrapping will be made at the firewall.
- Local users will be able to directly connect to the server, bypassing ProtoWrap's restrictions.

Disadvantages

- A firewall only protects a specific perimeter; any attack conducted from within the protected segment will talk directly with the server, not with the wrapper
- All logging is centralized on a single machine, so they can be easily erased by an attacker who gains access to the firewall
- Imposes extra load on the firewall, specially if a large number of servers is protected. If a firewall fails due to a DoS attack, the whole network will be disconnected from Internet.

This would be achieved using the following configuration files:

Firewall's `/etc/ipnat.conf` includes:

```
rdr ep1 192.168.0.1/32 port 25 -> 192.168.0.254 port 10025
```

Firewall's `/etc/rc.local` includes:

```
/usr/local/bin/smtpwrap &
```

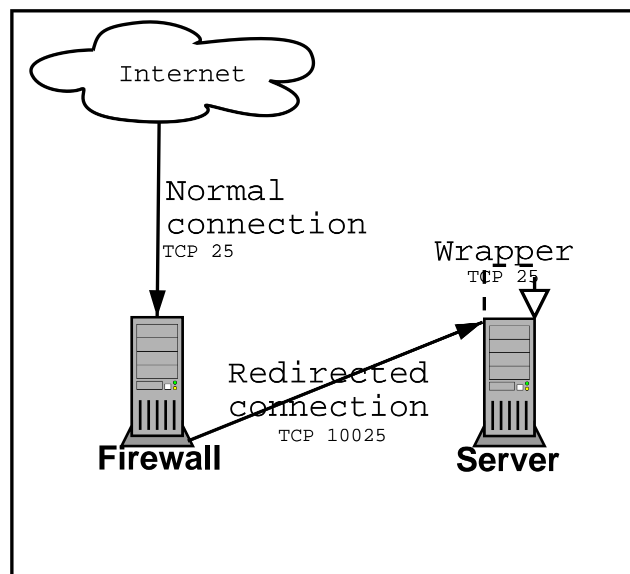
Firewall's `/usr/local/bin/smtpwrap`:

```
#!/usr/bin/perl

use ProtoWrap::SMTP;
use strict;

my ($wrap);

$wrap = ProtoWrap::SMTP->new('standalone' => 1,
                             'listenPort' => 10025,
                             'destType' => 'ip',
                             'destAddr' => '192.168.0.1',
                             'logLevel' => 3,
                             'maxMsgSize' => 3000000,
                             'relayDomainList' => ['mydomain.com'],
                             'maxRcpt' => 10,
                             'setUidTo' => 32767
                             );
die 'Can\'t start SMTP wrapper' if (not defined $wrap);
$wrap->startServer() or warn 'Can\'t start wrapper for '.$wrap->getProp();
```

5.2.2. Redirecting firewall

This scheme relies on the same basic ideas as the one presented in 5.2.1, but instead of running ProtoWrap at the firewall, each server machine runs a ProtoWrap

program. This program, however, does not run on the server's well-known port; it runs instead on an unused port. The firewall redirects the packages sent to the server to the port where ProtoWrap is running, and then ProtoWrap forwards them to the server.



From this scheme, we would get:

Advantages

- The servers' configuration must not be modified, the server programs will run as usual.

- Local users will be able to directly connect to the server, bypassing ProtoWrap's restrictions.
- Very useful while testing configurations — If a parameter sent to ProtoWrap is found to be wrong, the server can easily be directly accessed by modifying only one firewall rule.

Disadvantages

- A firewall only protects a specific perimeter; any attack conducted from within the protected segment will talk directly with the server, not with the wrapper
- Wrappers' executables and configuration are stored in each of the servers; this makes upgrading or adding rules a task requiring more effort than having them all in a single place.

The present configuration can be implemented this way:

Firewall's `/etc/ipnat.conf` includes:

```
rdr ep1 192.168.0.1/32 port 25 -> 192.168.0.1 port 10025
```

Server's `/etc/rc.local` includes:

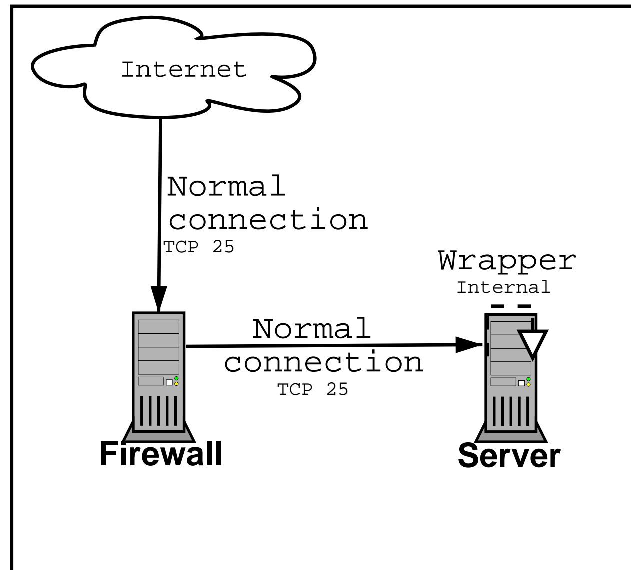
```
/usr/local/bin/smtpwrap &
```

Server's `/usr/local/bin/smtpwrap`:

```
#!/usr/bin/perl

use ProtoWrap::SMTP;
use strict;

my ($wrap);

$wrap = ProtoWrap::SMTP->new('standalone' => 1,
                             'listenPort' => 10025,
                             'destType' => 'ip',
                             'destAddr' => '127.0.0.1',
                             'logLevel' => 3,
                             'maxMsgSize' => 3000000,
                             'relayDomainList' => ['mydomain.com'],
                             'maxRcpt' => 10,
                             'setUidTo' => 32767
                             );
die 'Can\'t start SMTP wrapper' if (not defined $wrap);
$wrap->startServer() or warn 'Can\'t start wrapper for '.$wrap->getProp();
```

5.2.3. Server not running, called by the wrapper

This will be probably the most frequently used scheme. Here, the server processes (which are usually run from inetd) will not run at all. ProtoWrap will be installed in each individual server machine, and configured to wrap each individual server port. ProtoWrap can be run either in standalone or in inetd mode, and the server program will be called via a pipe, not via an IP connection. This way, the programs concurrently running at the server can be kept at a minimum, and network access to the protected services will be completely cut off.



For this scheme, we have:

Advantages

- Lower overall memory requirements
- Server programs are completely isolated from outsiders
- Does not require any usually awkward firewalling rules to be set
- Is not vulnerable to the local network, the same defense level is applied both to local and remote users

Disadvantages

- Higher initialization overhead for each incoming connection
- If many connections arrive simultaneously, memory usage can grow very fast
- Must modify the configuration for each server manually
- Will not work with servers that require interaction with a socket (IP connection), not with a file handle (STDIN)
- Wrappers' executables and configuration are stored in each of the servers; this makes upgrading or adding rules a task requiring more effort than having them all in a single place

Server's `/etc/rc.local` includes:

```
/usr/local/bin/smtpwrap &
```

Server's `/usr/local/bin/smtpwrap`:

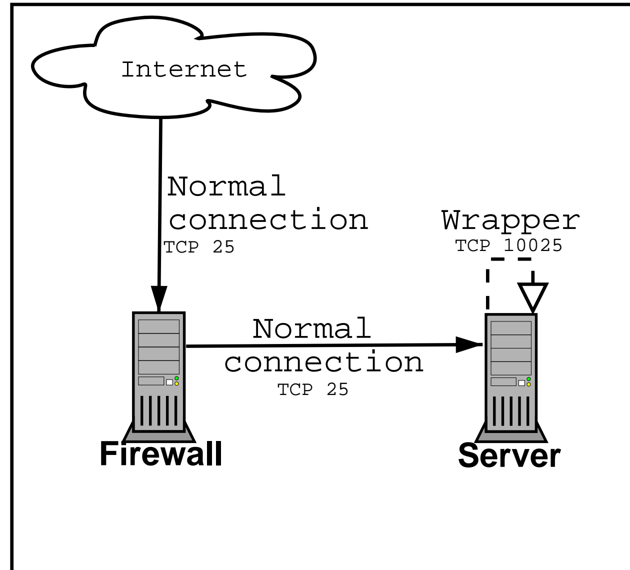
```
#!/usr/bin/perl

use ProtoWrap::SMTP;
use strict;

my ($wrap);

$wrap = ProtoWrap::SMTP->new('standalone' => 1,
                             'listenPort' => 10025,
```

```
                              'destType' => 'ip',
                              'destAddr' => '127.0.0.1',
                              'logLevel' => 3,
                              'maxMsgSize' => 3000000,
                              'relayDomainList' => ['mydomain.com'],
                              'maxRcpt' => 10,
                              'setUidTo' => 32767
                              );
die 'Can\'t start SMTP wrapper' if (not defined $wrap);
$wrap->startServer() or warn 'Can\'t start wrapper for '.$wrap->getProp();
```

5.2.4. Server running on a different port

The previous scheme had a serious limiting factor: Many server programs will insist on listening to a network port. However, they will almost always allow for relocation. In this scheme, the server's socket will be moved to a different port, and ProtoWrap will listen for communication on the server's original port. The author strongly reccomends adding local firewalling rules to avoid any direct external communication to the actual server's port.[6]

[6] Please note that, while the `ipchains` line in `rc.local` forbids any incoming connection to reach port 10025 on IP address 192.168.0.1, the wrapper's destination IP address is 127.0.0.1, which not only is easily accesible from within the server, not being affected by this rule, but also is inaccessible from any outside machine.



With this setup, we have:

Advantages

- Does not require any usually awkward firewalling rules to be set
- Very easy to troubleshoot, very easy to set up back to normal, unprotected operation if needed

Disadvantages

- The real server remains exposed to attacks unless a local firewalling rule is added

- Must modify the configuration for each server manually
- Wrappers' executables and configuration are stored in each of the servers; this makes upgrading or adding rules a task requiring more effort than having them all in a single place

To get this configuration, we would have:

Server's `/etc/rc.local` includes:

```
/sbin/ipchains -A input -d 192.168.0.1/32 --proto tcp --destination-port 10025 -j REJECT
/usr/local/bin/smtpwrap &
```

Server's `/etc/sendmail.cf` includes:

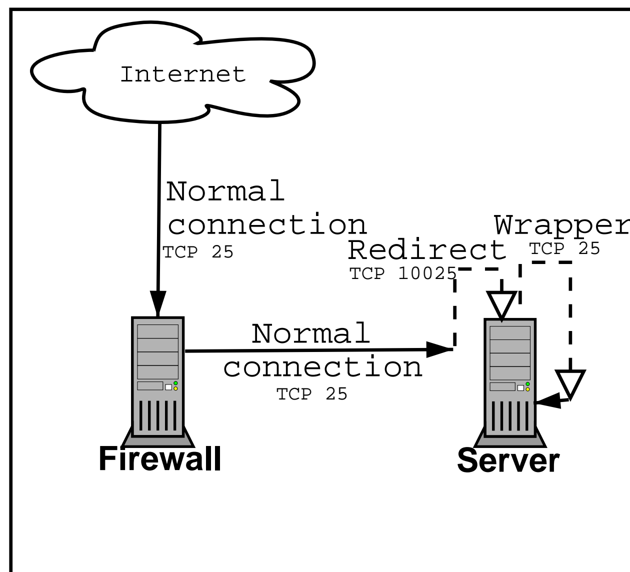
```
# SMTP daemon options
O DaemonPortOptions=Port=10025
```

Server's `/usr/local/bin/smtpwrap`:

```
#!/usr/bin/perl

use ProtoWrap::SMTP;
use strict;

my ($wrap);

$wrap = ProtoWrap::SMTP->new('standalone' => 1,
                             'listenPort' => 25,
                             'destType' => 'ip',
     'destAddr' => '127.0.0.1',
     'destPort' => 10025,
                             'logLevel' => 3,
                             'maxMsgSize' => 3000000,
                             'relayDomainList' => ['mydomain.com'],
                             'maxRcpt' => 10,
                             'setUidTo' => 32767
                             );
die 'Can\'t start SMTP wrapper' if (not defined $wrap);
$wrap->startServer() or warn 'Can\'t start wrapper for '.$wrap->getProp();
```

5.2.5. Local redirecting firewall local rules

There are cases, however, in which even the configuration change described above can not be attained. A server may be hard-wired at source code level to listen to a specific port, as many closed-source applications are, and it may not be desirable to run the wrapper at the firewall. Not all is lost, though, Using local firewalling rules, present in almost any Unix system, he can instruct the server to forward all incoming requests on the server's port to the wrapper's port, allowing only local connections (by local meaning originating in the same machine, not even network).



Advantages

- Server programs are completely isolated from outsiders
- Does not require the use of a third system as a firewall
- Is not vulnerable to the local network, the same defense level is applied both to local and remote users
- Server does not need to be reconfigured - It will recieve a completely normal-looking connection (of course, originating at localhost).

Disadvantages

- Wrappers' executables and configuration are stored in each of the servers; this makes upgrading or adding rules a task requiring more effort than having them all in a single place
- Writing and mantaining the local firewalling rules can be quite an awkward task

For this scheme, we would use the following files:

Server's `/etc/rc.local` includes:

```
/sbin/ipchains -A input -d 192.168.0.1/32 --proto tcp --destination-port 25 -j REDIRECT 10025
/usr/local/bin/smtpwrap &
```

Firewall's `/usr/local/bin/smtpwrap`:

```
#!/usr/bin/perl

use ProtoWrap::SMTP;
use strict;

my ($wrap);

$wrap = ProtoWrap::SMTP->new('standalone' => 1,
                             'listenPort' => 10025,
                             'destType' => 'ip',
                             'destAddr' =>  '127.0.0.1',
                             'logLevel' => 3,
                             'maxMsgSize' => 3000000,
                             'relayDomainList' => ['mydomain.com'],
                             'maxRcpt' => 10,
                             'setUidTo' => 32767
                             );
die 'Can\'t start SMTP wrapper' if (not defined $wrap);
$wrap->startServer() or warn 'Can\'t start wrapper for '.$wrap->getProp();
```

5.2.6. Other configurations

Of course, the configurations presented are only a proposal of what can be done. Many options have not been explored, such as a firewall with a scheme similar to 5.2.1 with many services wrapped, inetd-based wrappers, SMTP- and POP3-specific options, etc. However, with this section as an introduction it should be perfectly clear how to set up many different wrappers. Having all this different configuration posibilities makes ProtoWrap a very usable and convenient program, which, although presently is at an early, functioning stage, can soon evolve and become a very useful security tool.

A. ProtoWrap program code

A.1. Makefile.PL

```
use ExtUtils::MakeMaker;
WriteMakefile(NAME => "ProtoWrap",
              AUTHOR => "Gunnar Wolf",
              ABSTRACT => "Wrapper/validator for TCP connections",
              PM => {'ProtoWrap.pm' => '$(INST_LIBDIR)/ProtoWrap.pm',
                     'ProtoWrap/SMTP.pm' => '$(INST_LIBDIR)/ProtoWrap/SMTP.pm',
                     'ProtoWrap/POP3.pm' => '$(INST_LIBDIR)/ProtoWrap/POP3.pm'
                    },
              VERSION => 0.5
              );
```

A.2. ProtoWrap.pm

```
package ProtoWrap;

# CLASS ProtoWrap
#
# Class attributes:
```

```
#
# REQUIRED:
# listenPort - Port on which it should listen (if standalone
#              == 1)
# destAddr - Wrapped server's IP address (if destType eq 'ip')
# destPort - Wrapped server's destination port (if destType
#            eq 'ip')
# pipeCmd - Command to pipe communication to (if destType eq
#           'pipe')
#
# OPTIONAL:
# standalone - True if ProtoWrap will act in standalone mode.
#              False if it is to be called by inetd. Defaults to
#              true.
# destType - The type of connection to be opened to the server.
#            'ip' for IP connections, 'pipe' for pipes opened to
#             Unix commands. Defaults to 'ip'.
# maxLineLength - Maximum line length allowed. Setting it to
#                 zero disables checking. Defaults to 200.
# logLevel - Logging detail (0=none, 3=extra-verbose)
#            Defaults to 2.
# testLine - Reference to the function that will test each line
#            recieved from the client. If undefined, a default
#            testLine function is provided.
# testReply - Whether the data recieved from the server should
#             be checked (1) or only the data generated at the
#             client (0). Defaults to 0.
# logName - The service name to write to syslog with. If not
#           supplied, defaults to 'ProtoWrap'.
# setUidTo - Which user ID should the wrapper run as (numeric).
#            Defaults to current UID. Will not allow 0 (root).
# runSrvSuid - Whether to run the server program as root (1) or
#              to drop privileges just after acquired the
#              listening port (0). Defaults to 0.
#              Useful only if running with standalone=1 and
#              destType='pipe', ignored otherwise.
#
# GENERATED BY THE OBJECT:
# pid - PID of process handling the wrapper
# srcPort - Source port of the incomming connection
# srcIpAddr - Source IP address of the incomming connection
# srcName - Name of the remote host (resolved from srcIpAddr)

# Class public methods:
#
# new - creates a new ProtoWrap. Takes as argument a hash containing
#       the attributes and associated values for this wrapper
#       object.
#       If the object was not successfully created, returns undef.
# getProp - returns a hash with the wrapper object's attributes.
# set_maxLineLength - Modifies the maximum line length allowed for
```

```
#                     client to server communication
# set_logLevel - Modifies the detail of logging to be done
# startServer - Starts the wrapper server for this particular
#               instance
# stopServer - Stops the wrapper server for this particular
#              instance
# version - Returns ProtoWrap's version

use strict;
use IO::Handle;
use IO::Socket;
use IO::Select;
use IPC::Open2;
use POSIX qw(:sys_wait_h :signal_h);
use Sys::Syslog qw(:DEFAULT setlogsock);
$SIG{CHLD} = 'IGNORE';
$^W=1;

sub new {
    my $class = shift;
    my $self = { @_ };
    bless $self, $class;

    my (%needed,@optional,@destTypes,%userParams);
    # Test that all needed attributes are given, give the
    # default values to the optional values which were not
    # filled in, and check for the correct destination type
    # (IP or pipe).

    @destTypes = qw(ip pipe);
    $needed{ip} = ['destAddr', 'destPort'];
    $needed{pipe} = ['pipeCmd'];
    @optional = qw(standalone logLevel testLine maxLineLength
                   destType testReply logName setUidTo runSrvSuid);

    foreach (@optional) {
        if ($_ eq 'destType') {
            $self->{$_} = 'ip' unless defined $self->{$_};
        } elsif ($_ eq 'standalone') {
            $self->{$_} = 1 unless defined $self->{$_};
        } elsif ($_ eq 'logLevel') {
            $self->{$_} = 2 unless defined $self->{$_};
        } elsif ($_ eq 'testLine') {
            $self->{$_} = \&defaultTestLine unless defined $self->{$_};
        } elsif ($_ eq 'maxLineLength') {
            $self->{$_} = 200 unless defined $self->{$_};
        } elsif ($_ eq 'testReply') {
            $self->{$_} = 0 unless defined $self->{$_};
        } elsif ($_ eq 'logName') {
            $self->{$_} = 'ProtoWrap' unless defined $self->{$_};
        } elsif ($_ eq 'setUidTo') {
```

```
        $self->{$_} = $> unless defined $self->{$_};
    } elsif ($_ eq 'runSrvSuid') {
        $self->{$_} = 0 unless defined $self->{$_};
    } else {
        # We should never hit this spot...
        # If we do, we'd better die.
        die "Unexpected optional parameter $_";
    }
}

setlogsock('unix');
openlog($self->{logName},'cons,pid','ProtoWrap');

push (@{$needed{$self->{destType}}},'listenPort') if ($self->{standalone});

foreach (@{$needed{$self->{destType}}}) {
    if (not defined($self->{$_})) {
        $self->log (0,"$_ is a required parameter and was not supplied. ".
                    'ProtoWrap not started.');
        return undef;
    }
}

# Check if wrapper is being run as root. If so, refuse to
# continue if setUidTo is not set. We do not switch to a
# normal user yet - Root may be needed to listen on a low
# port or to run the server program.
if ($> == 0 && $self->{setUidTo} == 0) {
    $self->log (0,'Refusing to start - Running as root and setUidTo is not set');
    return undef;
}
# If a UID change is expected and we are not root, we
# will fail later on - It is better to fail early.
if ($> != $self->{setUidTo} && $> != 0) {
    $self->log (0,"Regular user (uid $>) will not be able to set UID to ".
                $self->{setUidTo});
    return undef;
}
# runSrvSuid is set to 0 (drop privileges as soon as possible
# unless standalone=1 and destType='pipe'
$self->{runSrvSuid} = 0 unless ($self->{standalone} == 1 &&
                                $self->{destType} eq 'pipe');

# We now check if there are any unexpected arguments
# supplied by the user.
# If so, we refuse to create the object and return undefined.
%userParams = $self->getProp;
foreach (@optional,@{$needed{$self->{destType}}}) {
    delete $userParams{$_};
}
if (keys(%userParams)) {
```

```
        $self->log (0,'Unexpected parameters: '.join (', ',keys(%userParams)).
                    ' - Refusing to continue.');
        return undef;
    }


    return $self;
}

sub DESTROY {
    my $self = shift;
    close $self->{listSock} if defined($self->{listSock});
}

sub set_maxLineLength {
    $_[0]->{maxLineLength} = $_[1];
}

sub set_logLevel {
    $_[0]->{logLevel} = $_[1];
}

sub getProp {
    return %{$_[0]};
}

sub startServer {
    my $self = shift;
    my ($error,$child);
    $error=0;
    if ($self->{standalone} == 1) {
        $self->{listSock} = IO::Socket::INET->new(
                                                    LocalPort => $self->{listenPort},
                                                    Type      => SOCK_STREAM,
                                                    Reuse     => 1,
                                                    Listen    => 10)
            or $self->log(0,"Couldn't be a TCP server on port $self->{listenPort}") &&
                ($error = 1);
        return undef if $error;
        $self->log(0,"Can't fork to accept incoming connections!: $@ $!") if
            (!defined($child=fork()));


        # If runSrvSuid is 0, we can drop privileges right now,
        # before any forking takes place
        # Check if UID change succeeded, and die otherwise.
        if ($self->{runSrvSuid} == 0) {
            $> = $self->{setUidTo};
            if ($> != $self->{setUidTo}) {
                $self->log(0,"UID change failed! UID is still $>, should be ".
                           $self->{setUidTo}.
```

```
                      " - Aborting execution.");
                die $self->{logName}.": UID change failed! UID is $>, should be ".
                    $self->{setUidTo};
            } else {
                $self->log(4,"UID changed - RUID: $<, EUID: $>, setUidTo ".
                           $self->{setUidTo});
            }
        }

        if ($child == 0) {
            # Child process - take care of the connection
            $self->getConn();
        } else {
            # Parent process - return the child process' PID
            $self->{pid} = $child;
        }
    } else {
        $self->getConn();
    }
}

sub stopServer {
    my $self=shift;
    $self->log(1,"Stopping server on port $self->{listenPort}");
    close $self->{listSock};
    kill ('TERM',$self->{pid});
    # We clear the PID attribute - The user-given attributes are
    # preserved in case the server is to be restarted or reused.
    delete $self->{pid};
    $self=undef;
}

sub getConn {
    my $self = shift;
    my ($cl_char, $src_conn, $child, $server_rd,$server_wr,
        $src_iaddr,$read_handles, $fromclient, $fromserver);

    while (my ($client_rd,$client_wr) = &getClient($self)) {
        if (!defined $client_rd || !defined $client_wr) {
            $self->log(0,'Error ocurred when setting up client connection - rd: '.
                       "$client_rd wr: $client_wr");
            next;
        }
        if ($self->{standalone} == 1) {
            # Standalone mode - Every recieved connection should get
            # its own server
            $self->log(0,"Can't fork to handle incoming client connection!: $@ $!") if
                (!defined ($child=fork()));
        } else {
            # inetd mode - The forking is done by inetd, so we act as
            # the child process
```

```
    $child = 0;
}
if ($child == 0) {
    # Child process - take care of the connection.
    if ($self->{standalone} == 1) {
        $src_conn=getpeername($client_rd);
        ($self->{srcPort},$src_iaddr)=unpack_sockaddr_in($src_conn);
        $self->{srcIpAddr} = inet_ntoa($src_iaddr);
        $self->{srcName} = gethostbyaddr($src_iaddr, AF_INET) ||
            '---unresolvable---';
        $self->log(1,'Connection recieved from '.$self->{srcName}.' ('.
                   $self->{srcIpAddr}.'), source port '.$self->{srcPort});
    }

    if ($self->{destType} eq 'ip') {
        # IO::Socket:INET->new opens a bidirectional socket,
        # so after assigning it to $server_rd, we assign
        # $server_rd to $server_wr
        $server_rd = IO::Socket::INET->new(
                                          PeerAddr => $self->{destAddr},
                                          PeerPort => $self->{destPort},
                                          Proto => 'tcp',
                                          Type => SOCK_STREAM) or die "$! - $@";
        $server_wr = $server_rd;
        $self->log(4,'Server answered');
    } elsif ($self->{destType} eq 'pipe') {
        my $cmdFile = $self->{pipeCmd};
        # Get only the command name, without any arguments
        $cmdFile =~ s/^(.+)\s.+$/$1/;
        if (-x $cmdFile) {
            # Clear any potentially dangerous enviroment variables
            $ENV{PATH} = '/bin:/usr/bin';
            $ENV{BASH_ENV} = '';
            ($server_rd,$server_wr) = (IO::Handle->new,IO::Handle->new);
            open2($server_rd,$server_wr,$self->{pipeCmd});
        } else {
            $self->log(0,$cmdFile . ' has invalid permissions (execution).');
        }
    } else {
        $self->log(0,'destType is not ip, is not pipe... What should I do with '.
                   $self->{destType}.'?');
    }

    # Server has started, we now set the UID to the low privilege
    # user (if running as root). Check if UID change succeeded,
    # and die otherwise.
    $> = $self->{setUidTo};

    if ($> != $self->{setUidTo}) {
        $self->log(0,"UID change failed! UID is still $>, should be ".
                   $self->{setUidTo}." - Aborting execution.");
```

```
    die $self->{logName}.": UID change failed! UID is $>, should be ".
        $self->{setUidTo};
} else {
    $self->log(4,"UID changed - RUID: $<, EUID: $>, setUidTo ".
               $self->{setUidTo});
}

# IO::Select allows us to listen to many different
# filehandles (in this case, sockets) at once. We
# will listen to $client_rd and $server_rd.
$read_handles = IO::Select->new($client_rd,$server_rd);

$self->log(4,"client: $client_rd $client_wr, server: $server_rd $server_wr");
while (1) {
    my @readable;
    @readable = $read_handles->can_read();
    $self->log(0,"Error in select $!") unless @readable;
    foreach my $sock (@readable) {
        my ($buff,$testResult);
        if ($sock eq $client_rd) {
            my $readed = sysread($client_rd,$buff,4096);
            unless($readed) {
                close $client_rd;
                close $client_wr;
                close $server_rd;
                close $server_wr;
                $self->log(3,'Closing communication with client '.
                           $self->{srcName}.' ('.$self->{srcIpAddr}.
                           ') port '.$self->{srcPort});
                exit 0;
            }
            $fromclient .= $buff;
            while((my $idx = index($fromclient,"\n")) != -1) {
                my ($cl_line,$tmp_cl_line);
                $cl_line = substr($fromclient,0,$idx+1,'');
                # $cl_line can be modified by the
                # testing function. $tmp_cl_line
                # should be used for logging.
                $tmp_cl_line = $cl_line;
                # We test the line. If the test
                # returns 0, we should not send
                # the line to the server. We pass the
                # client socket so error messages can
                # be reported back directly from the
                # routine.
                #
                # A negative return value indicates
                # the line was handled by the
                # routine, and we can ignore it.
                $testResult = $self->{testLine}->($self,\$cl_line,
                                                  $client_wr,0);
```

```
            if ($testResult > 0) {
                # Positive value - success
                #
                # \n at end of line could have
                # been chopped at testLine, so
                # append it if needed.
                $cl_line .= "\n" if (index($cl_line,"\n") == 1);
                $server_wr->print($cl_line);
                $self->log(5,"From client: $cl_line");
            } elsif ($testResult == 0) {
                # Zero returned - failure
                $self->log(1,'Line test failed');
                $self->log(2,"Offending line: $tmp_cl_line");
            } elsif ($testResult < 0) {
                # Negative value - handled by
                # test routine
                $self->log(3,"Handled by wrapper: $tmp_cl_line");
            }
        }
    } elsif ($sock eq $server_rd) {
        my $readed = sysread($server_rd,$buff,4096);
        unless($readed) {
            close $client_rd;
            close $client_wr;
            close $server_rd;
            close $server_wr;
            my $string = 'Closing communication with client';
            $string .= ' '.$self->{srcName} if defined $self->{srcName};
            $string .= ' ('.$self->{srcIpAddr}.')' if
                defined $self->{srcIpAddr};
            $string .= ' port '.$self->{srcPort} if
                defined $self->{srcPort};
            $self->log(2,$string);
            exit 0;
        }
        $fromserver .= $buff;
        while((my $idx = index($fromserver,"\n")) != -1) {
            my $sr_line = substr($fromserver,0,$idx+1,'');
            if ($self->{testReply}) {
                $testResult = $self->{testLine}->($self,\$sr_line,
                                                  $server_wr,1);
            }
            $client_wr->print($sr_line);
            $self->log(5,"From SERVER: $sr_line");
        }
    } else {
        $self->log(0,"Something happened - $sock not client nor server!");
        exit 0;
    }
}
```

```
            }
        } else {
            # Parent process - Clear $client_rd and $client_wr (they will
            # be handled only by the child process) and wait for more
            # incoming connections
            $client_rd = $client_wr = undef;
        }
    }
}

sub getClient {
    my ($self,$conn_rd,$conn_wr);
    $self=shift;
    if ($self->{standalone} == 1) {
        # Standalone mode - We wait for a connection and then return
        # two references to the connection (the same object, to be used
        # for reading and writing)
        $conn_rd = $conn_wr = $self->{listSock}->accept();
    } elsif ($self->{standalone} == 0) {
        # inetd mode - I/O will be carried using STDIN/STDOUT, so we
        # reference the filehandles.
        $|=1;
        $conn_rd = *STDIN;
        $conn_wr = *STDOUT;
    } else {
        # We should never hit this spot
        $self->log(0,'Invalid value for standalone: '.$self->{standalone}.
                   '. Unable to process connections.');
    }
    return ($conn_rd,$conn_wr);
}

sub defaultTestLine {
    # $line holds a reference to the line to be processed
    # $socket holds the socket from which the line came
    # $who indicates whether the line should be treated as a
    #      client-originated line (0, strict) or as a
    #      server-originated line (1, loose).
    # We return 0 if the test failed and we want ProtoWrap to issue an
    # error, a positive value if the test succeeded, and a
    # negative value if the answer was handled by the client and
    # we do not want ProtoWrap to pass it on.
    my($self,$line,$socket,$who) = @_;

    if (length($$line) > $self->{maxLineLength} && $self->{maxLineLength} > 0) {
        $$line = substr($$line,0,$self->{maxLineLength});
        $self->log(1,"Line too long\nChopping to:\n$$line");
    }
    return 1;
}
```

```
sub log {
    my($self,$level,$data,$appendTxt)=@_;
#    warn "$data\n" if $level <= $self->{logLevel};

    $data =~ s/[\r\n]//;
    $appendTxt = '- stage '.$self->{stage} if (defined $self->{stage});
    if ($level == 0) {
        syslog('err',"error: $data $appendTxt");
    } elsif ($level == 1) {
        syslog('info',"info: $data $appendTxt") if ($self->{logLevel} >= 1);
    } elsif ($level == 2) {
        syslog('notice',"notice: $data $appendTxt") if ($self->{logLevel} >= 2);
    } elsif ($level == 3) {
        syslog('debug',"debug: $data $appendTxt") if ($self->{logLevel} >= 3);
    } elsif ($level == 4) {
        syslog('debug',"debug: $data $appendTxt") if ($self->{logLevel} >= 4);
    } elsif ($level >= 5) {
        syslog('debug',"debug: $data $appendTxt") if ($self->{logLevel} >= 5);
    } else {
        # Better to be too verbose and not too terse
        # if unsure about log level desired
        syslog('debug',"debug: $data $appendTxt");
    }
}

sub version {
    return '0.5';
}

1;
```

A.3. ProtoWrap/SMTP.pm

```
package ProtoWrap::SMTP;
#
# ProtoWrap::SMTP - SMTP extension to ProtoWrap
#
use ProtoWrap;
@ISA = qw(ProtoWrap);
use strict;
use vars qw($crlf);

$crlf = chr(13).chr(10);

sub new {
    my ($class,$self,@smtpSpecific,@smtpNotProtoWrap,%tmpHash);
```

```
$class = shift;
$self = { @_ };
bless $self, $class;
# Parameters accepted for SMTP that are not part of the base
# ProtoWrap should be stored in a temporal hash until the object is
# created if we do not want ProtoWrap to complain, and then added
# to the existing object.
# The affected attributes are:
# maxMsgSize - the maximum message size allowed
# blockAddrList - A reference to an array with a list of regular
#   expressions representing blocked addresses.
# blockBodyList - A reference to an array with a list of regular
#   expressions representing lines to be blocked in the message body.
# maxRcpt - The maximum number of recipients specified with
#   RCPT TO:
# relayDomainList - List of domains which will be relayed - they are
#   required to appear either in the MAIL FROM or in the RCPT TO areas.
#   (regex anchored to end of string)
# relayIpList - List of IP addresses or ranges which will be accepted
#   for relay. (regex anchored to beginning of string)
@smtpNotProtoWrap = qw(maxMsgSize blockAddrList blockBodyList
                       relayDomainList relayIpList maxRcpt);

foreach (@smtpNotProtoWrap) {
    if (defined $self->{$_}) {
        $tmpHash{$_} = $self->{$_};
        delete $self->{$_};
    }
}

# Check for wrong parameters
@smtpSpecific = ('testLine','testReply');


foreach (@smtpSpecific) {
    if (defined $self->{$_}){
        $self->log(0,"Parameter $_ incompatible with SMTP");
        return undef
    }
}
# Set the SMTP defined parameters needed to create a ProtoWrap
if ($self->{destType} eq 'ip') {
    $self->{destPort} = 25 unless (defined $self->{destPort});
    $self->{destAddr} = '127.0.0.1' unless (defined $self->{destAddr});
} elsif ($self->{destType} eq 'pipe') {
    $self->{pipeCmd} = '/usr/lib/sendmail -bs' unless (defined $self->{pipeCmd});
}
$self->{standalone} = 1 unless defined $self->{standalone};
$self->{testLine} = \&defaultTestLine;
$self->{testReply} = 1;
$self->{logName} = 'ProtoWrap-SMTP';
```

```
    # Create the ProtoWrap. If it is undef, an error occured and we
    # should not start.
    $self = ProtoWrap->new(%$self);
    return undef if (not defined $self);
    # Add SMTP-specific attributes not allowed for base ProtoWraps,
    # and free %tmpHash's memory
    foreach (keys(%tmpHash)) {
        $self->{$_} = $tmpHash{$_};
        delete $tmpHash{$_};
    }
    # Set the needed parameters for a SMTP ProtoWrap that could not be
    # specified before creating a ProtoWrap
    $self->{maxMsgSize} = 10000000 if (not defined $self->{maxMsgSize});
    $self->{stage}=-1;
    # Make $self a SMTP object
    bless $self, $class;
}

sub defaultTestLine {
    my ($self,$line,$socket,$who) = @_;
    if ($who == 0) {
        return &fromClient($self,$line,$socket);
    } elsif ($who == 1) {
        return &fromServer($self,$line,$socket);
    } else {

    }

}

sub fromServer {
    my($self,$line,$socket,$who,$retValue);
    ($self,$line,$socket,$who) = @_;

    $retValue = 1;

    # We strip CR and LF to do the tests, add them back at the
    # end.
    $$line =~ s/[\r\n]//g;

    if ($self->{stage} == -1) {
        # Stage -1: Waiting for the server to say hello.
        # Instead of letting the real server expose its identity,
        # we give ours. We only check if the server's response
        # starts with 2, meaning that the incoming connection
        # should be accepted. We switch to stage 0, and signal that
        # we have still not decided to relay this message.
        $$line = '220 localhost.localdomain ESMTP ProtoWrap wrapper '.$self->version() if
            (substr($$line,0,1) eq '2');
        $self->{relayOk} = 0;
        # If the connection comes from an authorized relaying IP,
```

```
        # allow relaying.

        foreach (@{$self->{relayIpList}}) {
            $self->{relayOk} = 1 if (defined $self->{srcIpAddr} &&
                                    $self->{srcIpAddr} =~ /^$_/);
        }
        $self->{stage} = 0;
    } elsif ($self->{stage} == 1) {
        # If we get a 550 <address> Access denied message from
        # the server, and rcpts is 1, this means that the server
        # denied a sender address. We restore stage to 0.
        # If we are getting a 550 Access denied, we are almost
        # surely with 0 recipients... But double-checking does
        # not hurt.
        if ($$line =~ /^550 .+ Access denied/) {
            $self->{stage} = 0 if ($self->{rcpts} == 0)
        }
    }

    $$line .= chr(13).chr(10);
    return $retValue;
}

sub fromClient {
    # stage indicates the current stage of the connection, thus
    # indicating what will be accepted or rejected in the ongoing
    # communication.
    #
    my($self,$line,$socket,$who,$retValue);
    ($self,$line,$socket,$who) = @_;

    $retValue = 1;
    # We strip CR and LF to do the tests, add them back at
    # the end.
    $$line =~ s/[\r\n]//g;

    #
    # Our first test should be for maxLineLength
    #
    if (length($$line) > $self->{maxLineLength} && $self->{maxLineLength} > 0) {
        $$line = substr($$line,0,$self->{maxLineLength});
        $self->log(1,"Line too long\nChopping to:\n$$line");
    }

    # Stage -1 indicates that the server has not yet talked on
    # the connection, and all traffic before this should be
    # silently dropped.
    if ($self->{stage} < 0) {
        $$line = '';
        return -1;
    }
```

```
# If we are in stage 9, we should check against rejectMsg, maxMsgSize and
# any regexes found in blockBodyList
if ($self->{stage} == 9) {
    if ($$line eq '.') {
        if ($self->{msgLength} > $self->{maxMsgSize} && $self->maxMsgSize != 0) {
            # Report that message is too long and end the
            # connection. Closing the socket causes the server to
            # discard the message. We then exit cleanly.
            $socket->print('550 Message length exceeded - maximum allowed: '.
                           $self->{maxMsgSize}.' bytes'.$crlf);
            $socket->close;
            $self->log(1,'Terminating connection');
            exit 0;
        } elsif ($self->{rejectMsg}) {
            $socket->print('550 Message rejected - security check failed'.$crlf);
            $socket->close;
            $self->log(1,'Terminating connection');
            exit 0;
        } else {
            # Set everything up to start processing a new message
            $self->{stage} = 0;
            $self->{relayOk} = 0;
            foreach (@{$self->{relayIpList}}) {
                $self->{relayOk} = 1 if (defined $self->{srcIpAddr} &&
                                         $self->{srcIpAddr} =~ /^$_/);
            }
        }
    }
    if ($self->{rejectMsg}) {
        # Message has been flagged to be rejected. Pretend to continue
        # processing it. The server does not need to know about it,
        # anyway, as we are going to discard it.
        $$line = '';
        $retValue = -1;
    }

    # If we are over maxMsgSize, stop sending data to the
    # server
    if ($self->{msgLength} >= $self->{maxMsgSize}) {
        $$line = '';
        $retValue = -1;
    }
    # Check now against the blockBodyList
    foreach (@{$self->{blockBodyList}}) {
        if ($$line =~ /$_/i) {
            # If it matches, keep recieving the message (we can only
            # indicate failure after the message has been completely
            # recieved if we want to stay RFC compliant).
            $self->{rejectMsg} = 1;
            $self->log(1,"Message rejected - Matched blockBodyList rule $_");
```

```
        }
    }

    $$line .= chr(13).chr(10);
    $self->{msgLength} += length($$line);
    return $retValue;
}

# At this point, we are certain that we are in a command-only
# stage. SMTP Commands are all over four characters long.
if (length($$line) < 4) {
    $socket->print("500 Command unrecognized: \"$$line\"$crlf");
    $$line = '';
    $retValue = 0;

    # Commands which should be allowed no matter which stage
    # are we in.
} elsif (uc(substr($$line,0,4)) eq 'QUIT') {
    # If a QUIT is recieved, return only QUIT, chopping the
    # rest of the line.
    $$line = 'QUIT';
    $self->{stage} = 10;
} elsif (uc(substr($$line,0,4)) eq 'RSET') {
    # If a RSET is recieved, return only RSET, chopping the
    # rest of the line.
    $$line = 'RSET';
    $self->{stage} = 0;
    $self->{relayOk} = 0;
} elsif ($$line =~ /^HELO [\w\d\.\-\_]+/i || $$line =~ /EHLO [\w\d\.\-\_]+/i) {
    # Just allow it to reach the server. Warn if resolved name
    # is not similar to reported name
    my $reported = $$line;
    my $srcName = $self->{srcName};
    $reported =~ s/^(HE|EH)LO\s+//i;
    $self->log(4,"Resolved ($srcName) and reported ($reported) hostnames don't match")
        if (defined $srcName && defined $reported && $srcName !~ $reported
            && $reported !~ $srcName);

    # Convert EHLO into HELO - ProtoWrap wants the client to
    # stay in the base SMTP command set, disallowing
    # extensions
    $$line = 'HELO '.$reported;

    # Commands which should be disallowed or handled at the
    # wrapper no matter which stage are we in (should also
    # send an error message back to the client if adequate)
} elsif (uc(substr($$line,4)) =~ /^(VRFY|EXPN|SEND|SAML|SOML|TURN|HELP)/) {
    $socket->print("502 Sorry, we do not allow this operation$crlf");
    $$line = '';
    $retValue = 0;
} elsif (uc(substr($$line,0,4)) eq 'NOOP') {
```

```
        $socket->print("250 OK$crlf");
        $$line = '';
        $retValue = -1;
        # Now, commands specific to a certain stage
    } elsif ((not defined $self->{stage}) || $self->{stage} == 0) {
        # First stage,
        # Allow only MAIL FROM:<address>. While checking, delete
        # all unnecessary blank spaces.
        if ($$line =~ /^MAIL\s+FROM:/i) {
            # Is it a valid address?
            if ($$line =~
                s/^MAIL\s+FROM:\s*(\<*[\w\d\.\-\_\=]+\@[\w\d\.\-\_]*\>*)/MAIL FROM:$1/i) {
                # Sometimes addresses are supplied surrounded by angled
                # brackets <>. We can safely strip them out.
                $$line =~ s/[\<\>]//g;

                # Check against the blockAddrList and the relayDomainList
                my ($reject);
                $self->{msgFrom} = $$line;
                $reject = 0;
                $self->{msgFrom} =~ s/MAIL\s+FROM:\s*//i;
                foreach (@{$self->{blockAddrList}}) {
                    if ($self->{msgFrom} =~ /$_/i) {
                        $reject = 1;
                        $socket->print('550 '.$self->{msgFrom}."... Access denied$crlf");
                        $retValue = -1;
                        last;
                    }
                }
                foreach (@{$self->{relayDomainList}}) {
                    if ($self->{msgFrom} =~ /$_$/i) {
                        $self->{relayOk} = 1;
                    }
                }
                if ($reject == 0) {
                    $self->{stage} = 1;
                    $self->{rcpts} = 0;
                }
            } else {
                $retValue = 0;
                $$line =~ s/MAIL\s+FROM:\s*//;
                $socket->print("553 $$line... Domain name required$crlf");
            }
        } else {
            # Disallow everything else
            $retValue = 0;
            chomp $$line;
            $socket->print("500 Command unrecognized: \"$$line\"$crlf");
            $$line = '';
        }
```

```
} elsif ($self->{stage} == 1) {
    # Second stage.
    # Allow only RCPT TO:<address> and check if we have not
    # reached maxRcpt
    if ($$line =~
        s/^RCPT\s+TO:\s*(\<*[\w\d\.\-\_\=]+\@?[\w\d\.\-\_]*\>*)/RCPT TO:$1/i) {
        # Sometimes addresses are supplied surrounded by angled
        # brackets <>. We can safely strip them out.
        $$line =~ s/[\<\>]//g;

        # Allow relaying if no destination host is specified
        # (local user)
        $self->{relayOk} = 1 if (index($$line,'@') == -1);
        # relayOk is true if the original sender appears in the
        # relayDomainList
        if ($self->{relayOk} == 0) {
            # If it does not, check if the sender does
            $self->{msgTo} = $$line;
            $self->{msgTo} =~ s/RCPT\s+TO:\s*//i;
            foreach (@{$self->{relayDomainList}}) {
                if ($self->{msgTo} =~ /$_$/i) {
                    $self->{relayOk} = 1;
                }
            }
        }
        # If relayOk is still 0, the recipient should not be accepted
        if ($self->{relayOk} == 0) {
            $socket->print('550 '.$self->{msgTo}.'... Relaying denied'.$crlf);
            $$line = '';
            $self->log(1,'Relaying denied from '.$self->{msgFrom}.' to '.
                       $self->{msgTo}.
                       ' from  '.$self->{srcName}.'('.$self->{srcIpAddr}.')');
            $retValue = -1;
        } else {
            if ($self->{rcpts} < $self->{maxRcpt} || $self->{maxRcpt} == 0) {
                $self->{stage} = 2;
                $self->{rcpts}++;
            } else {
                $socket->print('551 Too many recipients ('.$self->{rcpts}.")$crlf");
                $self->log(1,'Too many recipients: '.$self->{rcpts});
                $$line = '';
                $retValue = -1;
            }
        }
    } else {
        # Disallow everything else
        $retValue = 0;
        chomp $$line;
        $socket->print("500 Command unrecognized: \"$$line\"$crlf");
        $$line = '';
    }
```

```
    } elsif ($self->{stage} == 2) {
        # Third stage
        # Allow DATA, switching to stage 9 and resetting the
        # message length counter, or allow for additional
        # RCPT TO, staying in stage 2.
        if (uc(substr($$line,0,4)) eq 'DATA') {
            $self->{stage} = 9;
            $$line="DATA";
            $self->{msgLength} = 0;
            $self->{rejectMsg} = 0;
        } elsif ($$line =~
                 s/^RCPT\s+TO:\s*(\<*[\w\d\.\-\_\=]+\@?[\w\d\.\-\_]*\>*)/RCPT TO:$1/i) {
            # Sometimes addresses are supplied surrounded by angled
            # brackets <>. We can safely strip them out.
            $$line =~ s/[\<\>]//g;

            if ($self->{rcpts} < $self->{maxRcpt} || $self->{maxRcpt} == 0) {
                $self->{stage} = 2;
                $self->{rcpts}++;
            } else {
                $socket->print('551 Too many recipients: '.$self->{rcpts}.")$crlf");
                $self->log(1,'Too many recipients ('.$self->{rcpts});
                $$line = '';
                $retValue = -1;
            }
        } else {
            # Disallow everything else
            $retValue = 0;
            chomp $$line;
            $socket->print("500 Command unrecognized: \"$$line\"$crlf");
            $$line = '';
        }

    } else {
        $$line = '';
        $retValue=0;
    }
    $$line .= chr(13).chr(10);
    return $retValue;
}

1;
```

A.4. ProtoWrap/POP3.pm

```
package ProtoWrap::POP3;
#
# ProtoWrap::POP3 - POP3 extension to ProtoWrap
#
use ProtoWrap;
@ISA = qw(ProtoWrap);
use strict;
use vars qw($crlf);

$crlf = chr(13).chr(10);

sub new {
    my ($class,$self,@pop3Specific,@pop3NotProtoWrap,%tmpHash);
    $class = shift;
    $self = { @_ };
    bless $self, $class;
    # Parameters accepted for POP3 that are not part of the base
    # ProtoWrap should be stored in a temporal hash until the object is
    # created if we do not want ProtoWrap to complain, and then added
    # to the existing object.
    # The affected attributes are:
    # maxLoginAttempts: The maximum login attempts permitted before
    # disconnection
    @pop3NotProtoWrap = ('maxLoginAttempts');

    foreach (@pop3NotProtoWrap) {
        if (defined $self->{$_}) {
            $tmpHash{$_} = $self->{$_};
            delete $self->{$_};
        }
    }

    # Check for wrong parameters
    @pop3Specific = ('testLine','testReply');


    foreach (@pop3Specific) {
        if (defined $self->{$_}){
            $self->log(0,"Parameter $_ incompatible with POP3");
            return undef
        }
    }
    # Set the POP3 defined parameters needed to create a ProtoWrap
    $self->{destType} = 'ip' if (!defined $self->{destType});
    if ($self->{destType} eq 'ip') {
        $self->{destPort} = 110 unless (defined $self->{destPort});
        $self->{destAddr} = '127.0.0.1' unless (defined $self->{destAddr});
    } elsif ($self->{destType} eq 'pipe') {
```

```
        $self->{pipeCmd} = '/usr/sbin/ipop3d' unless (defined $self->{pipeCmd});
    }
    $self->{standalone} = 1 unless defined $self->{standalone};
    $self->{testLine} = \&defaultTestLine;
    $self->{logName} = 'ProtoWrap-POP3';
    $self->{testReply} = 1;
    # Create the ProtoWrap. If it is undef, an error occured and we
    # should not start.
    $self = ProtoWrap->new(%$self);
    return undef if (not defined $self);
    # Add POP3-specific attributes not allowed for base PSNs,
    # and free %tmpHash's memory
    foreach (keys(%tmpHash)) {
        $self->{$_} = $tmpHash{$_};
        delete $tmpHash{$_};
    }
    # Set the needed parameters for a POP3 ProtoWrap that could not be
    # specified before creating a ProtoWrap
    $self->{maxLoginAttempts} = 0 if (not defined $self->{maxLoginAttempts});
    $self->{stage}=-1;
    # Make $self a POP3 object
    bless $self, $class;
}

sub defaultTestLine {
    my ($self,$line,$socket,$who) = @_;
    if ($who == 0) {
        return &fromClient($self,$line,$socket);
    } elsif ($who == 1) {
        return &fromServer($self,$line,$socket);
    } else {

    }

}

sub fromServer {
    # stage indicates the current stage of the connection, thus
    # indicating what will be accepted or rejected in the ongoing
    # communication. The relation between stage and the stages
    # defined in the RFC is:
    #
    # AUTHORIZATION:
    # 0 - USER has not yet been specified.
    # 1 - USER specified, waiting for server to answer.
    # 2 - USER has been specified, waiting for PASS.
    # 3 - USER and PASS specified, waiting for server's authorization
    # TRANSACTION:
    # 5 - Transaction state, all transaction commands are valid.
    #
    my($self,$line,$socket,$who,$retValue);
```

```
($self,$line,$socket,$who) = @_;

$retValue = 1;

# We strip CR and LF to do the tests, add them back at the
# end.
$$line =~ s/[\r\n]//g;

if ($self->{stage} == -1) {
    # Stage -1: Waiting for the server to say hello.
    # Instead of letting the real server expose its identity,
    # we give ours. We only check if the first character sent
    # by the server is '+', indicating success.
    $$line = '+OK POP3 localhost.localdomain ProtoWrap '.$self->version() if
        (substr($$line,0,1) eq '+');
    # We set the stage to 0 to allow for interaction to start
    $self->{stage} = 0;
    # As the connection is just starting, we set to zero all the
    # counters
    $self->{loginAttempts} = 0;
    $self->{maxMsgNumber} = 0;
    $self->{deletedMessages} = undef;
} elsif ($self->{stage} == 1) {
    # We do not care about the server's answer - USER should
    # always get a positive answer.
    $$line = '+OK User name accepted, password please';
    $self->{stage} = 2;
} elsif ($self->{stage} == 3) {
    if (substr($$line,0,1) eq '+') {
        $self->{stage} = 5;
    } else {
        $$line = '-ERR Bad login';
        $self->{stage} = 0;
        $self->{loginAttempts}++;
        if ($self->{maxLoginAttempts} && $self->{loginAttempts} >
            $self->{maxLoginAttempts}) {
            # Too many login attempts - Exit, terminating connection
            # automatically.
            $self->log(2,'Too many login attempts - terminating connection.');
            exit 0;
        }
    }
} else {
    # Checking how many messages are there waiting to be
    # retrieved. This is done right after the client sends a
    # STAT command.
    if (defined($self->{statJustAsked})) {
        # We should not go back here for the next line
        $self->{statJustAsked} = undef;
        # Get the relevant part of the answer into maxMsgNumber
        $self->{maxMsgNumber} = $$line;
```

```
            $self->{maxMsgNumber} =~ s/^.+(\d+).+$/$1/;
        }
    }

    $$line .= chr(13).chr(10);
    return $retValue;
}

sub fromClient {
    # stage indicates the current stage of the connection, thus
    # indicating what will be accepted or rejected in the ongoing
    # communication. The relation between stage and the stages
    # defined in the RFC is:
    #
    # AUTHORIZATION:
    # 0 - USER has not yet been specified.
    # 1 - USER specified, waiting for server to answer.
    # 2 - USER has been specified, waiting for PASS.
    # 3 - USER and PASS specified, waiting for server's authorization
    # TRANSACTION:
    # 5 - Transaction state, all transaction commands are valid.
    #
    my($self,$line,$socket,$who,$retValue);
    ($self,$line,$socket,$who) = @_;

    $retValue = 1;
    # We strip CR and LF to do the tests, add them back at
    # the end.
    $$line =~ s/[\r\n]//g;

    #
    # Our first test should be for maxLineLength
    #
    if (length($$line) > $self->{maxLineLength} && $self->{maxLineLength} > 0) {
        $$line = substr($$line,0,$self->{maxLineLength});
        $self->log(1,"Line too long\nChopping to:\n$$line");
    }

    # Commands that will always be handled the same way, no matter
    # where are we at
    if (uc(substr($$line,0,4)) eq 'QUIT') {
        # If a QUIT is recieved, return only QUIT, chopping the
        # rest of the line.
        $$line = 'QUIT';
        $self->{stage} = 10;
    } elsif (uc(substr($$line,0,4)) eq 'NOOP') {
        $socket->print("+OK This is your very special No-op!$crlf");
        $$line = '';
        $retValue = -1;
    } elsif (uc(substr($$line,0,4)) eq 'RPOP') {
        $socket->print("-ERR RPOP authorization not allowed$crlf");
```

```
    $$line = '';
    $retValue = 0;
} elsif ($self->{stage} < 0) {
# Stage -1 indicates that the server has not yet talked on
# the connection, and all traffic before this should be
# silently dropped.
    $$line = '';
    return -1;
} elsif ($self->{stage} == 0) {
    # Stage 0 should only allow, besides non-stage-specific
    # commands, USER.
    if (uc(substr($$line,0,4)) eq 'USER') {
        # check for format
        if ($$line =~ s/^USER\s+([\w\d\.\-\_]*)$/USER $1/i) {
            $self->{stage} = 1;
        } else {
            $$line = '';
            $socket->print("-ERR Invalid username $crlf");
            $retValue = -1;
        }
    } else {
        &invalid($line,\$retValue,$socket);
    }
} elsif ($self->{stage} == 2) {
    if (uc(substr($$line,0,4)) eq 'PASS') {
        # check for format
        if ($$line =~ s/^PASS\s+([\S]*)$/PASS $1/i) {
            $self->{stage} = 3;
        } else {
            $$line = '';
            $socket->print("-ERR Invalid password$crlf");
            $retValue = -1;
            $self->{loginAttempts}++;
            if ($self->{maxLoginAttempts} && $self->{loginAttempts} >
                $self->{maxLoginAttempts}) {
                # Too many login attempts - Exit, terminating connection
                # automatically.
                $self->log(2,'Too many login attempts - terminating connection.');
                exit 0;
            } else {
                &invalid($line,\$retValue,$socket);
            }
        }
    }
} elsif ($self->{stage} == 5) {
    if (uc(substr($$line,0,4)) eq 'RSET') {
        # Send only RSET, ignoring the rest of the line
        $$line = 'RSET';
        # Reset the deleted message list
        $self->{deletedMessages} = undef;
    } elsif (uc(substr($$line,0,4)) eq 'LAST') {
```

```
    # Send only  LAST, ignoring the rest of the line
    $$line = 'LAST';
} elsif (uc(substr($$line,0,4)) eq 'STAT') {
    # Send only  STAT, ignoring the rest of the line
    $$line = 'STAT';
    $self->{statJustAsked} = 1;
} elsif (uc(substr($$line,0,4)) eq 'LIST') {
    # Send only  LIST, ignoring the rest of the line
    $$line = 'LIST';
} elsif (uc(substr($$line,0,4)) eq 'RETR' || uc(substr($$line,0,3)) eq 'TOP') {
    # Store in $msgNum just the message number requested
    my $msgNum = $$line;
    my $lines = $$line;
    if (uc(substr($$line,0,4)) eq 'RETR') {
        $msgNum =~ s/^RETR\s*(\d+).*$/$1/i;
    } else {
        $msgNum =~ s/^TOP\s*(\d+).*$/$1/i;
        $lines =~ s/^TOP\s*\d+\s*(\d+).*$/$1/i;
    }
    # Check if it has already been deleted
    if (defined($self->{deletedMessages}->{$msgNum})) {
        $$line = '';
        $retValue = -1;
        $socket->print("-ERR Marked as deleted.$crlf");
    } elsif ($msgNum > $self->{maxMsgNumber} || $self->{maxMsgNumber} == 0) {
        # Message number greater than the total number of messages
        $$line = '';
        $retValue = -1;
        $socket->print("-ERR Invalid message number.$crlf");
    } else {
        # Ok, go ahead...
        if (uc(substr($$line,0,4)) eq 'RETR') {
            $$line = "RETR $msgNum";
        } else {
            $$line = "TOP $msgNum $lines";
        }
    }
} elsif (uc(substr($$line,0,4)) eq 'DELE') {
    # Store in $msgNum just the message number requested
    my $msgNum = $$line;
    $msgNum =~ s/^DELE\s+(\d+).+$/$1/i;
    # Check if it has already been deleted
    if (defined($self->{deletedMessages}->{$msgNum})) {
        $$line = '';
        $retValue = -1;
        $socket->print('-ERR Message already deleted.$crlf');
    } else {
        $$line = "DELE $msgNum";
        $self->{deletedMessages}->{$msgNum} = 1;
    }
} else {
```

```
            # Disallow everything else
            &invalid($line,\$retValue,$socket);
        }
    } else {
        # Guess the line was recieved when it is the server's turn to
        # talk. Respond with an error and ignore it.
        &invalid($line,\$retValue,$socket);
    }

    $$line .= chr(13).chr(10);
    return $retValue;
}

sub invalid {
    my ($line,$retValue,$socket) = @_;
    $socket->print("-ERR Invalid command$crlf");
    $$line = '';
    $$retValue = 0;
}

1;
```

Bibliography

[1] de Raadt, Theo. (1998). Lecture: *"Secure Code Auditing"*. Dictated at computo.98@mx congress, UNAM, México D.F.

[2] *Stackguard.* (1998-2000). *"Immunix: Adaptive system viability"*. <http://www.immunix.org> .

[3] Venema, Wietse. (1992). *"TCP WRAPPER Network monitoring, access control, and booby traps"*. Mathematics and Computing Science Eindhoven University of Technology. <ftp://ftp.porcupine.org/pub/securitytcp_wrapper.txt.Z>.

[4] Ranum, Marcus. (1996). "Smap - SMTP queuer". Trusted Information Systems. <http://www.fwtk.org/fwtk/docs/mjr-slides/>.

[5] Aquino, Ruben. Torres, Olga Lidia (1999). *"Sendmail 8.10.1 con reglas ANTI-SPAM"*. Computer Security Department tutorials, DGSCA-UNAM. <http://www.asc.unam.mx/Tutoriales/Tutoriales/sendmail/index.html>.

[6] Postel, Jonathan. (1982). *"Simple Mail Transfer Protocol"*. Internet Engineering Task Force — Request For Comments. RFC 821. <http://www.ietf.org/rfc/rfc0821.txt?number=821>.

[7] M. T. Rose. (1988). *"Post Office Protocol - Version 3"*. Internet Engineering Task Force — Request For Comments. RFC 1081. <http://www.ietf.org/rfc/rfc0821.txt?number=1081>.

[8] *"OpenSSL: The Open Source toolkit for SSL/TLS"*. (2000). The OpenSSL Project. <http://www.openssl.org>.

[9] *"SSLWrap"*. (1999). The SSLWrapper project. <http://www.rickk.com/sslwrap/>.

[10] Schwartz, Randall. Christiansen, Tom. Wall, Larry. (1997) *Learning Perl* (2nd ed.). O'Reilly & Associates. (ISBN 1-56592-284-0).

[11] Wall, Larry. Christiansen, Tom. Schwartz, Randall. (1996) *Programming Perl* (2nd ed.). O'Reilly & Associates (ISBN 1-56592-149-6).

[12] Srinivasan, Sriram. (1997). *Advanced Perl Programming*. O'Reilly & Associates (ISBN 1-56592-220-4).