

ALIN-ADRIAN ANTON

SECURE SMTP PROXY FOR PROTECTING
MAIL TRANSFER AGENTS

Scientific Coordinator
Prof.Dr.Eng.Crețu Vladimir-Ioan

SECURE SMTP PROXY FOR PROTECTING MAIL TRANSFER AGENTS

ALIN-ADRIAN ANTON



University Degree Diploma
Department of Computers
Faculty of Computer Engineering
Polytechnic University of Timișoara

June 2007

*Alin-Adrian Anton: Secure SMTP Proxy for Protecting Mail
Transfer Agents, University Degree Diploma, © June 2007*

[May 22, 2007 at 8:07]

*These airplanes we have today are no more than a perfection of a
child's toy made of paper.*

— Henri Coandă

Dedicated to humans.

To all the medics who assist birth.

To all the families who assist them.

ABSTRACT

Security solutions are usually generic and exhaustive. Only classic attacks and a small set of vulnerabilities can be covered by generic approaches. More, the electronic mail system is the oldest of the services provided by the Internet. Research has been seriously approached only around lower-level threats like viruses and unsolicited commercials. Serious threats like *network penetration* simply lack the proper investigation which may take the service's peculiarities into account and develop a particularized solution. Heterogenous protection schemes can do better, but still the situation is not satisfactory.

I am trying to raise the flag and show that the oldest and most important Internet service deserves better attention and can be better protected due to its particular nature and mechanisms.

DESCRIEREA CONȚINUTULUI

Soluțiile curente de securitate sunt preponderent generice și exhaustive. Doar atacurile comune pot fi abordate prin soluții generale. Deși poșta electronică este unul din primele servicii oferite de Internet, cercetarea în domeniul securității acesteia este orientată spre vulnerabilități cu risc mai redus, precum reclamele nesolicitate și virușii. Amenințările serioase - precum accesul neautorizat - duc lipsă de o cercetare elaborioasă care să pună accent pe mecanismele interne ale sistemului de poștă electronică.

Voi încerca să ridic ștafeta și să demonstrez că cel mai vechi și mai important serviciu Internet poate fi protejat mult mai bine dacă sunt studiate particularitățile sale în-time.

*Nothing is more difficult, and therefore more precious, than to be
able to decide.*
— *Napoleon Bonaparte*

ACKNOWLEDGEMENTS

I wish to acknowledge my scientific coordinator, Prof. Dr. Eng. Crețu Vladimir for the patience and efforts granted towards my passion.

Without your encouragements and open support my journey would have not been possible. Thank you for supporting my opensource quests.

Many thanks to everybody who manages to stay around me, or doesn't have a choice!

CONTENTS

PART I THEORETICAL SKETCH	1
1 INTRODUCTION	3
1.1 Current state	3
1.1.1 General Overview	4
1.1.2 StackGuard	5
1.1.3 ProPolice	6
1.1.4 Randomized Stack Space	7
1.1.5 Randomized Heap Space	8
1.1.6 Boundary check patch for GCC	8
1.1.7 Non-executable stack space	9
1.1.8 Intrusion Detection Systems	10
1.1.9 Conclusion	10
1.2 Content description	11
PART II THE SHOWCASE	13
2 BASIC KNOWLEDGE	15
2.1 Store and Forward Systems	15
2.2 The SMTP Protocol	16
2.2.1 HELO Command	16
2.2.2 MAIL Command	17
2.2.3 RCPT Command	17
2.2.4 DATA Command	17
2.2.5 RSET Command	18
2.2.6 NOOP Command	18
2.2.7 QUIT Command	19
2.2.8 Error codes	19
2.3 Extensions to the protocol	21
2.3.1 The EXPN Command	21
2.3.2 The VRFY Command	22
2.4 Type of threats	22
2.4.1 Structure of the e-mail system	22
2.4.2 Unsolicited commercials	24
2.4.3 Viral worms	24
2.4.4 Backdoor threats	24
2.4.5 Denial of Service	25
2.4.6 Intrusion	25

3	PROBLEM SPECIFICATION	27
3.1	SMTP Syntax Verifier	28
3.2	Realtime Shellcode Disassembler	28
4	DESIGN CONCEPTS	31
4.1	General Design Criteria	31
4.2	Modular structure	31
4.2.1	Security	32
4.2.2	Stability	33
4.3	Expandability	33
4.4	Ergonomy	33
4.5	Performance	34
5	IMPLEMENTATION	35
5.1	How it Works	35
5.2	Modularity	37
5.2.1	config.h	37
5.2.2	cmdline.c	38
5.2.3	daemon.c	39
5.2.4	modules.c	42
5.2.5	mylog.c	44
5.2.6	packet.h	45
5.2.7	registry.c	46
5.2.8	rpx.c	47
5.2.9	subnet.c	48
5.3	Security	50
5.4	Expandability	51
5.5	Ergonomy	52
5.6	Sample Filter Modules	52
5.6.1	SMTP Syntax Verifier	52
5.6.2	Realtime Shellcode Disassembler	54
6	EVALUATION	61
6.1	Benchmarking	61
6.2	System requirements	63
6.3	Source code complexity	64
7	CONCLUSION	67
7.1	Future work	67
	BIBLIOGRAPHY	69

LIST OF FIGURES

Figure 1	Store and Forward System	15
Figure 2	Mail Transfer Agent Logic Blocks	22
Figure 3	The proxy guards the network border	35
Figure 4	Internal multi-threaded operations detail	36
Figure 5	Plug and Go Technology - unlimited filter modules	52
Figure 6	Throughput comparison	63
Figure 7	Performance penalty variation	63

LIST OF TABLES

Table 1	SMTP Error Codes	20
Table 2	Source Code Organization	37
Table 3	Benchmark with proxy	61
Table 4	Benchmark without proxy	62
Table 5	Proxy complexity metrics	64
Table 6	Syntax module complexity metrics	64
Table 7	Disassembler module complexity metrics	65

ACRONYMS

ASCII	American Standard Code for Information Interchange
BLIP	Basic Loop Integer Protection
BNF	Backus-Naur Form

CVE	Common Vulnerabilities and Exposures
ESMTP	Extended Simple Mail Transfer Protocol
IDS	Intrusion Detection System
LMTP	Local Mail Transport Protocol
MDA	Mail Delivery Agent
MTA	Mail Transfer Agent
MTP	Mail Transfer Protocol
RFC	Request For Comments
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol

Part I

THEORETICAL SKETCH

INTRODUCTION

1.1 CURRENT STATE

A new era was born when humans started to use computers as communication tools. Geographic borders vanished, in spite of the fact that the telegraph and international phone lines existed long before.

Evolving fast, the electronic mail system is one of the oldest service of the Internet. Simply put, it is probably the first concrete digital encapsulation of the meaning and the goal: humans sharing information.

A few decades ago, in 1972¹, Ray Tomlinson² first used the @ symbol to separate the username from the hostname, and thus established the syntax of the e-mail address. Great ideas like that last in time, and Internet pioneer Jon Postel is credited to calling it a "nice hack" - it certainly has lasted to this day.

Some trace e-mail origins back to 1965, in form of the MAILBOX system used at MIT...

Originally published in August 1982 the Simple Mail Transfer Protocol (SMTP) standard as defined by RFC821³ is still in use today - though certain extensions were attached, to resolve the new needs.

Ongoing research needs to be done in order to elaborate on the new challenges that evolution provides, developing the system and preserving the principles that governed its birth: openness and freedom of speech.

¹ <http://www.livinginternet.com>

² <http://www.nethistory.info>

³ <ftp://ftp.rfc-editor.org>

1.1.1 General Overview

Reaching maturity fast, there are plenty of protection solutions surrounding Mail Transfer Agent (MTA) systems; most of them deal with viruses, worms or unsolicited commercials.

Genuinely the oldest Internet service, the e-mail system also tends to be the most popular. A simple web site may be backed up by multiple MTA servers, some for incoming, and some for outgoing messages.

None of these merits help mail servers benefit from the same level of intrusion protection as other services. Security in conjunction with e-mails usually implies client-side protection from Internet malware.

Half of the vulnerabilities related to MTA can be remotely abused. At the Common Vulnerabilities and Exposures (CVE) center 57⁴ out of 136 MTA programming bugs provide possible vectors for a remote system compromise.

No specific solution for MTA intrusion protection is available...

Dates go back to 1999, from the time of this writing. Arguably the most complete database of vulnerabilities, it just lists the well known and commonly met issues.

Flaws are difficult to analyze because international corporations still tend to minimize the impact of bad publicity when it comes to severe bug exposures. Information may so be less accurate, and in many cases a *proof of concept* scenario is unavailable.

Greatly inspired by the product of *TrustWall HTTP Proxy*⁵, the current work tries to re-implement the idea and adapt it to MTA technology.

At best there are generic solutions which apply to operating system scale and work for any userland service...

Having this said, following there are major technologies with individual approaches to the problem of intrusion prevention. They are generally applicable from mail servers

⁴ Almost half of them can be remotely triggered

⁵ <http://freshmeat.net/projects/twhttpd/>

to any other kind of service one can imagine, so they are not [MTA](#) specific but can be used to protect [MTA](#) servers.

1.1.2 *StackGuard*

As described by [Wagle and Cowan](#) [8], StackGuard is a technology which does not literally prevent stack overflows; it attempts to detect them before they become dangerous.

Limited to preventing the execution flow of being redirected to user controlled memory addresses, StackGuard claims no guarantee that memory corruption is detected all the time. Linear adjacent locations can still be overwritten and depending on their original contents and how they are interpreted by the conditional branches in the code, takeovers can still take place.

With StackGuard, canary values are inserted inside the stack frames of function calls, so that when the saved return address is corrupted they are destroyed - this provides a mechanism to prove that a stack overflow took place and the real function will no longer return at the user supplied address. A special mechanism is triggered and the normal execution flow is blocked from it's course (the process is terminated).

Normally execution flow hijacking is achieved by overwriting the saved return address in a stack activation frame with user supplied values, so when the function returns it will execute the code from a user supplied location which, by all means, may contain foreign instructions.

There are variations of the same idea which may include the randomization of canary values, read-only canaries and other improvements. Each converges to the same point: when the function returns from the call, the very first thing which needs to be done is to check the authenticity of the canary value.

Directed to classic stack overflows as introduced by the notorious [Levy](#) [5] article, StackGuard is just a part of a

large puzzle and cannot guarantee immunity - it's just an improvement.

The problem is that if an opponent knows, is able to precompute or even guesses the canary values, the whole mechanism is rendered useless.

1.1.3 *ProPolice*

Originally known as SSP from [Etoh and Yoda \[2\]](#), ProPolice is an improvement of StackGuard. By using random canary values and complementing stack integrity checks with variables relocation[4] it enhances the safety in the stack overflow domain.

Expanding the addressed problem set, the two IBM researchers came in with the idea of re-arranging the stack frame so that it will converge to an "ideal" model. Brilliantly, the array variables are always located at the highest part of the stack frame, such that when they become overflowed no local variables are affected.

Organizing function arguments in respect to the C calling convention implies that they are stored "above" the local regions of the stack frame. So in order to achieve protection, they are copied into locally assigned variables - safe from the growing side of any possible buffer.

Safety-model functions do not always work. Both a pointer to a function and a character array encapsulated into a C structure will still expose the pointer when the array is overflowed. Random changes of the order of elements inside a structure is prohibited and should not be part of the conversion.

Again, dynamic arrays cannot be protected by compiler conversions and optimizations. Instant changes can appear during run-time execution. Normally, [MTAs](#) mostly use dynamic allocations for storing user-supplied arrays.

1.1.4 Randomized Stack Space

Stack space is organized by the kernel. A subroutine loads the executable process into memory, and prepares the required resources. In order to randomize the stack space addresses a compiler patch is not enough. Dynamically loadable kernel modules or even kernel patches are required.

It's tight dependency with the operating system makes this method difficult to port. Ruling out different hardware architectures, it is even hard to port across similar operating systems running on the same hardware platform.

Ultimately, once implemented and activated, any process running on the host will benefit of the protection with a simple restart. Lately even the kernel itself becomes protected by the patch.

Execve() system call is usually modified to allocate more space than it is necessary, then place the stack anywhere in the wasteful gap.

*Twinkle, twinkle
little star...*

Again, this doesn't prevent stack overflows at all. Notably it becomes considerably difficult for someone to overflow a certain desired address in the stack, bearing in mind that the beginning of the stack is unknown. Yet, the smaller the gap, the likely the guess.

Within 32-bit or smaller size systems, as [Shacham et al. \[6\]](#) showed, stack space randomization is anything but magic. A determined opponent will manage to "guess" whatever is needed and get over it.

Years have to pass before hardware developers will provide serious mechanisms for address space randomization - if they ever will. Limitations like high production costs or the necessary waste of system resources may prove this idea to be unfeasible on real world scenarios - especially where performance is a demand.

Unlike office-styled workstations, heavy loaded servers will never afford to dump memory and execution cycles to the recycle bin.

1.1.5 *Randomized Heap Space*

Not very different than the previous idea, the same principle is applied to *malloc()*. Generally, the same issues and debates can be brought into discussion against heap address randomization. Sadly, however, the penalties may be more substantial as the heap is far larger than the stack and is dynamically modified at higher rates, especially on server machines.

...how I wonder
where you are...

A server constantly forces the heap to grow and decrease. It is even difficult to prevent the heap space from address fragmentation and performance must be traded with storage capacity.

Daemons, and especially [MTA](#) may allocate and deallocate large chunks of data during long runtime sessions - as long as decent e-mail attachments are allowed.

In a truly randomized version of the heap space, allocating and deallocating 10 Megabytes of data is very problematic. Hereby fragmentation of memory appears in a blink of an eye, and the whole system becomes unusable for any real purpose.

1.1.6 *Boundary check patch for GCC*

A completely different approach than those already discussed, the Basic Loop Integer Protection ([BLIP](#)) patch introduced by [Horovitz](#) [3] addresses the issue of integer bugs.

Very often the patch is simply inefficient regarding overflows, unless integers are used in some algebraic computation to determine the size or position in a buffer.

Everyday use may imply a combined solution of compiler

patches and operating system extensions, a mixed collaboration of technologies able to prevent most of the bugs and scenarios.

Anyhow, some doors will remain opened and some solutions may not fit the production environment, so there is still room for the danger to come in.

Irresponsibly mixing all of the possible compiler patches and enhancements will not suffice. Remaining there are still bugs inside the protected perimeter, and the deceiving feeling of safety may appear.

1.1.7 *Non-executable stack space*

Hardware platforms treat executable accesses differently. Each code execution cycle will generate a *memory read* access and the architectural details may vary from vendor to vendor. Alpha and SPARC processors separate[7] *memory reads* from *code fetching* cycles so there is full hardware support for implementing non-executable stack space.

Rudimentary support is also offered by the x86 architecture, and NetBSD⁶ (which is probably the most portable operating system at the date of this writing) has a working implementation already tested on production servers.

The trick is that wherever the hardware allows it, the *mmap()* system call provides a *PROTEXEC* permission bit which denotes if the memory zone is executable or not. Stack zones can be marked as read/write only and even if someone tries to execute processor instructions from it, a system trap will be triggered and the code will fail to run.

Adjacent buffers can still be overflowed, and if they contain sensitive information - like passwords and other type of credentials - it is still possible to circumvent authentication mechanisms.

⁶ <http://www.netbsd.org/>

1.1.8 Intrusion Detection Systems

Intrusion detection systems are passive; they cannot prevent intrusion. Detecting intrusion allows for extensive information logging and deploying certain configurable actions, but it may provide a subtle weakness in the whole defensive system: like fire alarms, they ring when smoke is detected. It is impossible to detect real smoke without having sources of fire - so the alarm rings when something inside the protected perimeter is already burning.

Probably a bad alarm is better than no alarm; this usually translates into sending an e-mail and adding the offending IP address to the firewall blacklist. Unsuspecting targets may be completely disabled if the "missile" is intelligent enough to take out the protection mechanisms surrounding the impact zone.

Many Intrusion Detection System (IDS)s just monitor and scan the traffic searching for specific signatures, exactly as antiviruses inspect a file. Protection can only be partially achieved, and it is very easy to fall into a false sense of security. Because of that, Snort⁷ (the most popular opensource IDS) can be interconnected with various complementary software modules and it is designed for interoperability.

1.1.9 Conclusion

Linux and other unix-like operating systems provide generic and exhaustive protection schemes in the form of security patches or addons. Ordinary attacks can be blocked and detected, but that is far from enough.

On the other hand, MTA systems require *performance*, *robustness* and *scalability*, all of which are very prohibitive for most of the current solutions. Developers should concentrate efforts in order to find a specific solution.

*Current solutions
are generic and
insufficient...*

⁷ <http://www.snort.org>

1.2 CONTENT DESCRIPTION

Security solutions are usually generic and exhaustive. Only classic attacks and a small set of vulnerabilities can be covered by generic approaches. More, the electronic mail system is the oldest of the services provided by the Internet. Research has been seriously approached only around lower-level threats like viruses and unsolicited commercials. Serious threats like *network penetration* simply lack the proper investigation which may take the service's peculiarities into account and develop a particularized solution. Heterogenous protection schemes can do better, but still the situation is not satisfactory.

I am trying to raise the flag and show that the oldest and most important Internet service deserves better attention and can be better protected due to its particular nature and mechanisms.

Part II

THE SHOWCASE

BASIC KNOWLEDGE

2.1 STORE AND FORWARD SYSTEMS

There are plenty of events which may interrupt the mail transportation process or force it to fail. Whatever node inside the followed path encounters a reboot, a hardware failure or a power outage, the system should be able to adapt.

A store and forward system is a way to implement fault tolerance regarding mail transportation. No message should be lost, even if the errors are permanent.

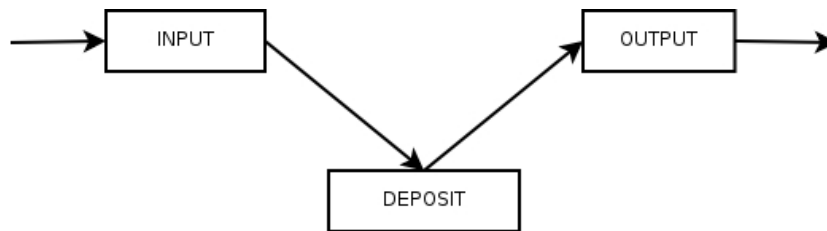


Figure 1. Store and Forward System

The problematic message is queued in a local deposit for later retries, or until another route can be planned for the final destination. Everytime unnecessary delays occur, the sender recipient can be informed. Delivery really fails only after a few days on most unix servers.

The message still gets deposited and delayed for a small amount of time (usually a matter of seconds) even if the destination is straightforward ready. Oftenly this allows for process and privilege separation inside the [MTA](#) implementation. One process can do queue management, another one route planning and different levels can be used for the

transportation problem. All this shall be detailed later.

Last but not least, how does the system know when the remote failure is permanent or temporary? Looking at the [SMTP](#) protocol and its error code compression algorithm should be self-explanatory.

2.2 THE SMTP PROTOCOL

September 1980 - the Mail Transfer Protocol ([MTP](#)) was first formalized by RFC772¹. Almost one year later, in May 1981 the author² updates it by RFC780. It was a mixture of *telnet*³ and *ftp*⁴ commands so not very elegant.

Derived from [MTP](#), the *Simple Mail Transport Protocol* was first published in RFC788⁵ and updated by RFC821 one year later. Notably, this milestone version lasted for two decades and it's going to be referred by this document.

Only 7 commands are demanded for a minimalistic RFC compliant implementation, and they should be understood by any receiver. Syntaxes are extracted in Backus-Naur Form ([BNF](#)) notation from the original publication:

2.2.1 HELO Command

Opening of a transmission channel implies that the caller advertises its domain name:

```
HELO <SPACE> <domain> <CR> <LF>
```

Hostnames usually comply to the following regular expression, tested and designed during production use:

¹ <http://www.ietf.org>

² Jonathan B. Postel

³ RFC764

⁴ RFC765

⁵ November 1981

```
#define RXP_DOMAIN:
```

```
(([:alnum:])+)|((([:alnum:])+|
[:alnum:]+\-[[:alnum:]]+)\.)+[:alpha:]+)
```

Example:

```
HELO basement.localdomain
250 mail.yahu.com
```

2.2.2 MAIL Command

```
MAIL <SPACE> FROM: <address> <CR> <LF>
```

Sender and recipient addresses conform to the following regular expression (again, this was polished from trial and errors during production used):

```
#define ADDRESS:
```

```
(([:alnum:]]+(@RXP_DOMAIN))
```

There can be only one sender per message. A *RXP_DOMAIN* should translate into the previous definition.

2.2.3 RCPT Command

Yet, the *RCPT* command may be invoked multiple times, as one sender can transmit for an unlimited number of recipients during one single session:

```
RCPT <SPACE> TO: <address> <CR> <LF>
```

whereas address conforms to the previously defined syntax.

2.2.4 DATA Command

```
DATA <CR> <LF>
```

Each body section begins with a *DATA* command. Data buffers are converted to printable American Standard Code

for Information Interchange ([ASCII](#)) codes (where applicable) and sent on the wire just as they are (including white spaces from the message) until the terminating empty line signal. It is not obvious but the terminating line may take different forms:

<CR> <LF> <DOT> <CR> <LF>

if the line is not also the first, or:

<DOT> <CR> <LF> if the message body is empty.

Nevertheless a [MTA](#) cannot differentiate between valid or invalid mail buffers; as long as the protocol is not violated, the real contents of a message are out of it's scope. Later the contents can be forwarded to specialized scanners which may determine if it's potentially dangerous or invalid.

2.2.5 *RSET Command*

RSET <CR> <LF>

A *reset* clears out the state machine in the remote server, provoking instant amnesia for everything that was learned in the current incomplete transmission⁶.

2.2.6 *NOOP Command*

NOOP <CR> <LF>

Transmitting dummy no-operation commands keeps the network link open by refreshing delay counters on both sides. Eventually usefull on long transmissions over low quality⁷ links.

⁶ An incomplete transmission is when the end of message DOT was not yet transmitted.

⁷ With bandwidth bottlenecks or packet loss.

2.2.7 *QUIT Command*

QUIT <CR> <LF>

Rude and unconforming client implementations simply omit this *"goodbye"* procedure, by closing the transmission channel and thus breaking the rules.

Postfix⁸ attempts to reply with a funny error message whenever it has the chance to "punish" rule breakers:

```
telnet somedomain.com 25
Trying 193.X.X.X...
Connected to somedomain.com.
Escape character is '^]'.
220 somedomain.com ESMTP Postfix
GET / 1.1
221 Error: I can break rules too. Goodbye.
Connection closed by remote host.
```

instead of:

QUIT

221 Bye.

2.2.8 *Error codes*

An error code is composed from three digits. Important digits have higher decimal ranks, while less significant digits are the rightmost. Numerical values for each digit represent an index in an imaginary error table - so the rightmost digits encode the details while the leftmost signal major general states.

Critical errors start with the major digit set to 5. Others decrease in value in proportion with their severity. Normal situations begin with a major of 2, temporary errors with 4.

⁸ <http://www.postfix.org/>

211	System status, or system help reply
214	Help message
220	Service ready
221	Service closing transmission channel
250	Requested mail action okay, completed
251	User not local; will forward to o3@some.net
354	Start mail input; end with . (a dot)
421	Service shutting down NOW
450	Requested mail action not taken: mailbox full (try later)
451	Requested action aborted: local error in processing (try later)
452	Requested action not taken: insufficient system storage
500	Syntax error, command not understood
501	Syntax error in parameters or arguments
502	Command not implemented
503	Bad sequence of commands
504	Command parameter not implemented
550	Requested action not taken: no such user here
551	User not local; please try root@upt.ro
552	Requested mail action aborted: /tmp full?
553	Requested action not taken: mailbox name not allowed
554	Transaction failed

Table 1. SMTP Error Codes

Very practical and elegant, the second digit converges the problem to a more specific issue and the last digit, where not null, points to an exact match in the virtual error table.

It is so very easy to translate the error messages into any human readable language whenever new definitions are added. No additional source code needs to be developed.

Critical errors are considered permanent. Everything else is temporary and due to its *fault tolerant* design the client system can try again later.

2.3 EXTENSIONS TO THE PROTOCOL

Due to the evolution of the Internet and new communication needs, in 2001⁹ the standard was updated with RFC2821.

Simple [SMTP](#) is enough for the transmission of messages. It can handle any kind of e-mail message. Nothing has been changed in the new version, only some extensions have been incrementally added.

Compatibility is retained by the HELO/EHLO version handshake. Extended transmission channels are opened by the *EHLO* command while normal channels continue to use the *HELO* version. They both share the same syntax as it was described before.

2.3.1 *The EXPN Command*

EXPN <SPACE> <address>

Having this extension enabled on the mail server may pose a security threat. Expansion of local user names may be a policy violation or may even provide sensitive details to an outside peeker.

Normally the expansion is only allowed to specific private networks, most of the times located in the same geographical area - like different buildings and local offices.

A network where usernames take the form of:

aa37553@cs.utt.ro

defenately needs such an extension enabled.

⁹ April 2001

2.3.2 The VRFY Command

VRFY <SPACE> <address>

Not very useful in the current context - and possibly unsafe - the command only verifies if an address is known by the remote server party. Yet it is present on most distributions at the date of this writing, and sometimes it's even abused by unsolicited commercial senders to harvest new addresses for their database.

2.4 TYPE OF THREATS

2.4.1 Structure of the e-mail system

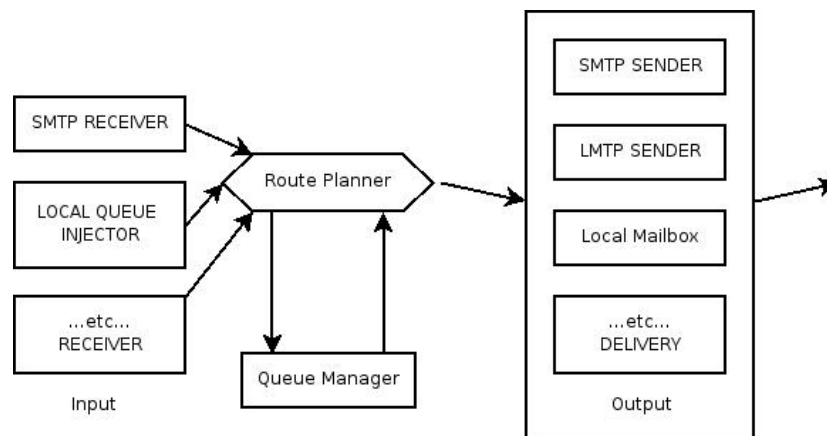


Figure 2. Mail Transfer Agent Logic Blocks

SMTP Receiver

Main interface and impact zone between the [MTA](#) and the Internet. Receives and handles incoming transmission channels in parallel.

Local Queue Injector

Sends local mail messages into the spool database. Handy for debugging or web scripting.

The Route Planner

Inspects the recipients for the messages passing through the temporary deposit, and determines one or more routes to be taken so that destination is reached.

Queue Manager / Spool Manager

The reliability of the queue manager makes the system fault tolerant and robust. Carries an interface to the local deposit of e-mails.

SMTP Sender

A simple and internal [SMTP](#) client which delivers mail to the next network hop - only used when the local queue injector is not appropriate.

Local Mail Transport Protocol ([LMTP](#)) is a simplification of the [SMTP](#) protocol, and is sometimes used for local delivery of the messages - for example through an AntiVirus scanner which resides on the same machine.

Local Mailbox

New messages are stored (in an atomic manner) into the local user's mailbox file. Because there are many disputed file formats and sometimes complex databases are implied, the actual delivery is performed by a Mail Delivery Agent ([MDA](#)) component which guarantees for atomic operations.

Exposure to attacks is concentrated around the *receiver interface* and thus my efforts try to isolate a solution for protecting that component of any generic [MTA](#) system.

2.4.2 *Unsolicited commercials*

Best known as "spam", it is probably the most commonly met and damaging attack to what used to be the first free service of a free Internet. Our days programmers are easily destroying the concepts and principles of the mail service by taking radical measures against this type of threat.

Some popular DNS blacklists will help you not receive mail from about 40% of the Internet. Some call this a solution - and some will ask you to pay for it as a service.

Most threats do not target the system itself, but the reader's client...

Eventually the real epicentre is located on the client side, and unconsenting users get their Inbox full of garbage or even so hard to search that they constantly need to use a new address. Once on target, the address remains permanently listed on the offender's database.

2.4.3 *Viral worms*

Freedom is never free, and computer viruses are just a notorious sample price. Worms are the new evolution of computer viruses, aimed with the ability to propagate and infect remote machines over the Internet. There are a lot of commercial *cat-and-mouse* races for this issue and the current solutions mostly manage to do the job.

However a lot of expensive network bandwidth is wasted and mail system administration increases in complexity and price.

This attack is also targeted against the e-mail client side, having no major impact against the [MTA](#) level.

2.4.4 *Backdoor threats*

Backdoors are customized computer worms which are ment to take over the client side one way or the other, when the mail is opened.

Some even exploit the ignorance and computer illiteracy of the users, and thus bring the client side computers into illegal activities such as sending commercials without prior subscription.

2.4.5 Denial of Service

Anyone anywhere is vulnerable to bandwidth exhaustion. This is unrelated to the services or the size of the network in question. MTAs are not an exception to this.

Currently there is no better solution then the close colaboration between major Internet service providers.

2.4.6 Intrusion

Injecting binary code into a foreign process is clearly demonstrated by Levy[5] in his now notorious *PHRACK Magazine*¹⁰ article.

In short, computer programs use the stack. Mostly for depositing function arguments and variables. But the compiler uses the stack too, when it translates the sources into executable code.

Intrusion always has a clearly defined target and scenario...

There is no real delimiter between the stack used by the compiler, and the stack used by the program. That makes collision possible. Software errors may produce stack space overlaps between the compiler and the program, and when the overlap is carefully controlled by a user of the application, it can result in a total takeover of the target.

There are many advanced techniques for injecting machine code into the program's execution flow. Some are applicable to the stack, some use the heap[7], and methodologies may vary with the target architecture and operating system.

¹⁰ <http://www.phrack.org/>

However it is certain that for execution to take place, the binary object code needs to be passed to the program via one or more of its arguments. Thus a working solution has to detect and block the binary object code before it reaches the target, and this must be done by inspecting the program's arguments.

PROBLEM SPECIFICATION

Intrusion seems to be the most devastating scenario from all above. Giving that, how can it be stopped?

Not many Internet services are *store-and-forward* systems. Only a few can really benefit from the peculiarity that they don't need to be instant!

Rather than being time-dependant, the e-mail system was not designed to be realtime or blazingly fast. A user will expect the mail to travel the World instantly (i.e. in a matter of seconds) - however, e-mails may take up to a few days to arrive at the final destination, just like real world postcards.

Nevertheless, why wait so much if it's already possible to have real-time communication services, like voice over ip, text chats, video conferences etc. ?

Classic post office mail works like this. Every snail mail implies that you go to the post office, send a postcard, the agency makes sure it travels the right paths so that it will reach destination - and then, when your recipient is at home and has the time, will check the Inbox and find the message.

It's the way real mail works, and it has been exploited in the electronic system in a way that provides fault and defect tolerance. Services are able to find a path across the "world" even if there are temporary power line outages or even permanent defects to intermediary networks - this is what the store and forward system is all about.

Before anyone ever had the idea, anti-virus software already took advantage of this system particularity. Later "anti-spam"¹ filters learned to exploit it even more ingeniously, by introducing intentional adaptive delays inside the [SMTP](#)

¹ The "spam" term is really borrowed from an old Monty Python sketch.

protocol.

It could be exploited by intrusion protection filters too, and if it's going to work it should open fertile horizons for defeating [MTA](#) level intrusion.

Such a dangerous approach may put the whole theory at stake, but the benefits are certainly worth the challenge. Sample software is attached and tested on real world scenarios, with two *proof of concept* modules.

3.1 SMTP SYNTAX VERIFIER

The first one guards against violation of [SMTP](#) syntax. It is based on regular expressions and attempts to be much more restrictive than the internal code of the [MTA](#), without breaking the Request For Comments ([RFC](#)).

For instance, most [MTAs](#) won't check if there are unprintable character sets inside the input traffic. This is because programming a mature syntax checker is not trivial and takes time. The syntax checker provided by this proxy was long tested on production servers and it can be considered very mature and stable.

Almost all common attack scenarios make use of unprintable character sets inside the program's arguments, or at least break the strict [RFC](#) syntax in one way. These scenarios can be easily detected and solved by the syntax checker module.

The implementation will be detailed later.

3.2 REALTIME SHELLCODE DISASSEMBLER

The beauty of the "beast" is the approach of detecting dangerous executable operands inside the mail traffic.

The *shellcode* term is a shortcut from "executable operand code which provides shell terminal access". That is, operands which translate into correct system calls like *execve("/bin/sh")*

and if interpreted by the target processor are able to grant system access to an unauthorized person.

Not all shellcodes do that, but most of them do. So shellcode is a generic term for a stream of foreign processor instructions which are injected into the application's execution flow.

Most detectors around the world, key-mechanisms inside any decent IDS, are just string comparison functions with relatively large signature sets.

They are all vulnerable to new, unknown and possibly complex shellcodes which may try hard to circumvent the detection. In the wild, there are samples of pure alphanumeric attacks for different processor architectures (some time ago, they were just a myth).

What this proxy does is attempting to check, in real time, if parts of the input buffer can be understood by the target processor as intelligible operand code. It simply tries to disassemble parts of the input buffer, to see if there are decompilation errors or not, and then attempts to interpret the decompiled operands for further analysis.

This approach is not possible in time critical services, like web browsing. However, electronic mail is not time critical - and that is the key.

DESIGN CONCEPTS

4.1 GENERAL DESIGN CRITERIA

The software is a security proxy, so first of all, it has to be *secure*.

Performance is a must, since it is necessary to handle heavy loads of parallel mail traffic. The proxy stands in front of the [MTA](#), so it must not degenerate all the work done by talented programmers to optimize the throughput.

Being in front of the [MTA](#), the proxy becomes mission critical. If the proxy fails to deliver, all the users behind it will have their Inbox disabled. This can turn into a disaster, so *stability* is very important.

It should be very easy for other developers to contribute and expand the software, so *expandability* must be taken into account right from the beginning.

At last, system administrators are mostly lazy persons and prefer *ergonomic* solutions with least possible administration complexity.

4.2 MODULAR STRUCTURE

Writing security-critical software introduces a standalone programming paradigm, which differentiates itself from all normal procedures of writing "good" software.

Security code must be as clear as possible. Lisibility is probably the most important goal. One word: if you can't read it, DON'T!

So how to achieve lisibility?

As opposed to object oriented programming, lisibility of security code deals with *how* things are implemented, not *what* services are provided.

Writing secure code is very difficult. This is one strong reason why safety code is always small. Being small, it also is crystal clear and streightforward (as long as related to the implementation).

It should be noted that by *secure programming* the author reffers to "how to implement", and does not cover design mistakes like weak protocols, poor algorithms, etc. Secure programming is one paradigm, and designing secure protocols is something completely different.

Any security mechanism (including software) can be viewed as a secure protocol which implies four steps:

1. authentication
2. identification
3. audit
4. authorization

However these are not covered by the subject, as what the proxy does is replace insecure [MTA](#) implementations with secure watchdogs.

4.2.1 *Security*

Everywhere and all around the programming paradigms, modularity improves lisibility. Organized source code is easier to understand and read.

Therefore, modularity is a must when implementing secure software.

However writing secure software is a state of the art and many of the principles are better described by Wheeler [9] in his notorious *Linuxdoc HOWTO*¹.

4.2.2 Stability

Stability (robustness) of the software is very important. Stable programs are not always secure... but secure programs should always be stable.

Robustness is a direct achievement of clear lisible code. And as secure software should be crystally clear lisible, stability becomes a metric of security.

4.3 EXPANDABILITY

Modular source code is expandable source code. However things can be improved even more, with a *plug-and-go*² mechanism which allows modules to be independently attached or removed during runtime.

This is what every new programmer who wishes to contribute is going to appreciate most.

4.4 ERGONOMY

Everyone knows most system administrators are lazy guys, including myself. So for a software to be popular, it should be easy to use and require - if possible - no administration.

"Good" software from sysadmin's perspective is something you easily install, and forget about it for decades. One might even forget it's there (it happened..). So this must be taken into account, if there's an honest desire to improve the safety of MTAs.

¹ <http://www.linuxdoc.org/>

² Will be described in detail in the Implementation chapter.

4.5 PERFORMANCE

Writing secure software is usually done in the C language. This means that if the source code is grouped into modules, the compiled binary code is not - as opposed to object oriented compilers.

Performance therefore suffers no penalty loss because of the modular approach.

However the supplementary communication links and security checks will most certainly introduce some performance penalties. If these penalties are too severe, the whole effort may be rendered useless.

IMPLEMENTATION

The provided source code is an elegant *Proof of Concept* daemon which demonstrates that the suggested approaches are reliable solutions to [MTA](#) intrusion attacks.

The regular expressions used by the syntax module are very mature and were tested on production environments.

The disassembler module is based on Libdasm¹, a C library which is able to parse Intel x86 operands and translate them into assembly routines. A proof of concept x86 emulator is used for detecting malicious operations.

5.1 HOW IT WORKS

The proxy is a Layer-7 firewall² and guards the network perimeter from possible attacks.

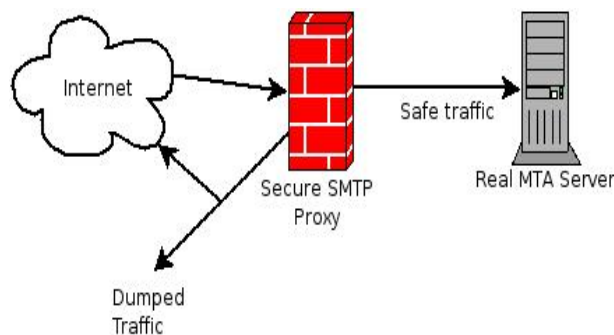


Figure 3. The proxy guards the network border

Threads instantiation is faster than a call to *fork()* since there is less data to copy and allocate. That is why unix

¹ <http://www.nologin.net/>

² Application layer in OSI reference model

threads were chosen for *multitalking* inside the system. The next figure uses curves for multithreaded communication links, and all are bidirectional channels. The straight arrows symbol thread creation and single-threaded activities.

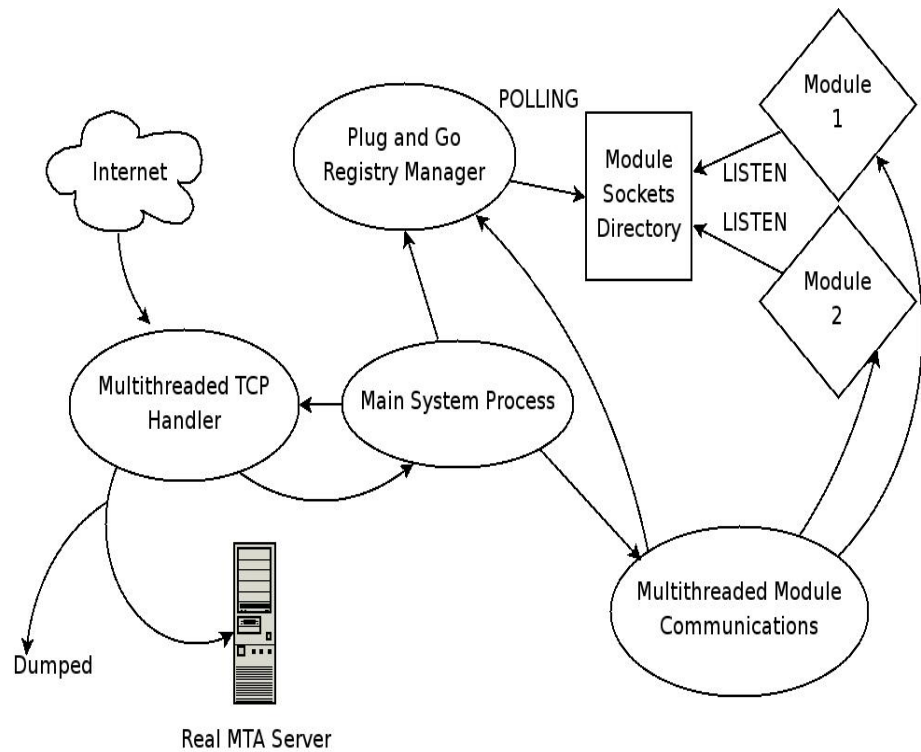


Figure 4. Internal multi-threaded operations detail

The rectangular modules are filters which can be plugged in and out during runtime, without interrupting the service. They are decisional filters which have the intelligence to allow or reject buffer streams.

Only buffers allowed by all the active filtering modules will be forwarded to the real [MTA](#). Everything else is discarded and the sender informed.

5.2 MODULARITY

The daemon source code is divided into organized files:

config.h	Holds configuration variables and knobs
cmdline.c	Parses command line input for daemon configuration
daemon.c	Transmission Control Protocol (TCP) handler
modules.c	Communicates with the independent attached filters
mylog.c	A file logging facility
packet.h	Protocol used by filters talking to the base proxy
registry.c	Registry of active filters, <i>plug-and-go</i> mechanisms
rxp.c	Regular expressions compiler and tester
security.c	Sandbox with least possible privileges
subnet.c	Utility functions for IPv4 addresses and netmasks
testall.c	The <i>main()</i> module which starts the process

Table 2. Source Code Organization

Essential implementation details for important modules are presented below:

5.2.1 *config.h*

```
#include <sys/types.h>

#define PROGNAME "DIPLOMA"
#define VERSION  "PoC-1"
#define MAX_POSIX_LINE 2048

/* total number of module sockets */
#define MAX_SOCKETS 50

/* total number of concurrent TCP connections */
#define MAX_CONN_CNT 25

#define LOG_FILENAME "audit.log"
#define SOCKET_DIR "/socks/"
```

10

```

/* directory polling interval */
#define SLEEP_SEC 5

/* TCP inactivity patience */
#define TCP_SEC_TIMEOUT 60

int    sandbox_gid;
int    sandbox_uid;
int    local_port;
int    server_port;
uint32_t local_ip;
uint32_t server_ip;

char sandbox_path[MAX_POSIX_LINE];

```

The source itself is explanatory. Lisibility was very important and taken into account.

5.2.2 *cmdline.c*

The command line parser makes use of the standard unix *stdlib* library for elegantly handling the start-up options.

```

#define LOCAL_ADDRESS 1
#define LOCAL_PORT    2
#define SERVER_ADDRESS 5
#define SERVER_PORT    6
#define SANDBOX_PATH  7

static struct option longopts[] = {
    { "local_address",    required_argument, NULL, LOCAL_ADDRESS},
    { "local_port",      required_argument, NULL, LOCAL_PORT},
    { "server_address",  required_argument, NULL, SERVER_ADDRESS},
    { "server_port",     required_argument, NULL, SERVER_PORT},
    { "sandbox_path",    required_argument, NULL, SANDBOX_PATH},
    { NULL,              0,              NULL, 0}
};

int
_usage(char *myname)
{
    fprintf(stderr, "Usage: \n %s [options]\n", myname);
    fprintf(stderr, "Options are:\n");
    fprintf(stderr, "    --local_address 10.0.0.1\n");
    fprintf(stderr, "    --local_port 25\n");
    fprintf(stderr, "    --server_address 10.0.0.10\n");
}

```

```

fprintf(stderr, "    --server_port 25\n");
fprintf(stderr, "    --sandbox_path /usr/local/nobody/\n");
fprintf(stderr, "\n\n");

return (-1);
}

```

30

Some of them are optional, some are necessary:

```

if (local_ip == 0) {
    mylog(MYLOG_ERR, "cmdline_preload(): local_ip not set");

    return (-1);
}

if (server_ip == 0) {
    mylog(MYLOG_ERR, "cmdline_preload(): server_ip not set");

    return (-1);
}

if (strncmp(sandbox_path, "lazy_bastard", strlen("lazy_bastard")) == 0) {
    mylog(MYLOG_ERR, "cmdline_preload(): sandbox_path not set");

    return (-1);
}

```

10

Wherever string functions are used, the *safe alternatives* are preferred and care is taken to correctly limit the buffer lengths.

5.2.3 *daemon.c*

Most importantly takes care of the traffic transit between the server and the Internet:

```

int
_socket_transit(int insock, int outsock)
{
    struct timeval tv;
    struct packet *pkt;
    fd_set readfds, savedfds;
    char buf[MAX_POSIX_LINE];
    int ret;
}

```

```

tv.tv_sec = TCP_SEC_TIMEOUT;
tv.tv_usec = 0;

FD_ZERO(&savedfds);
FD_SET(insock, &savedfds);
FD_SET(outsock, &savedfds);

for (;;) {
    readfds = savedfds;
    if ((ret=select(FD_SETSIZE, &readfds, NULL, NULL, &tv)) == -1) {
        mylog(MYLOG_ERR,
            "_socket_transit(): select(): %s", strerror(errno));

        close(insock);
        close(outsock);

        return (-1);
    }

    if (ret == 0) {
        mylog(MYLOG_INFO,
            "_socket_transit(): timeout on select(), closing session");
        ...
    }

    if (FD_ISSET(insock, &readfds)) {

        /* there is data from Internet connection */

        memset(buf, 0x0, MAX_POSIX_LINE);
        if ((ret=recv(insock, buf, MAX_POSIX_LINE - 1, 0)) == -1) {
            mylog(MYLOG_ERR, "_socket_transit(): recv(): %s",
                strerror(errno));
            ...
            return (-1);
        }
        ...

        /* here modules are called to test the safety of buf */

        if ((pkt=check_safety_modules(buf)) == NULL) {
            mylog(MYLOG_ERR,
                "_socket_transit(): packet allocation failed");

            return (-1);
        }
    }
}

```

```

switch (pkt->status) {
case STATUS_OK:
    if ((ret=send(outsock, buf, strlen(buf), 0)) == -1) {
        mylog(MYLOG_ERR,
            "_socket_transit(): send(): %s", strerror(errno));
        ...
        return (-1);
    }
    if (ret < strlen(buf)) {
        mylog(MYLOG_ERR, ...);
    }
break;
case STATUS_ERROR:
    mylog(MYLOG_ERR, ...);
break;

/* attacks should be logged individually by the modules */

case STATUS_ATTACK:
    ...
break;
} /* switch() */
free(pkt); /* do not waste RAM */
}
if (FD_ISSET(outsock, &readfds)) {

    /* there is some data from SMTP server */

    memset(buf, 0x0, MAX_POSIX_LINE);
    if ((ret=recv(outsock, buf, MAX_POSIX_LINE - 1, 0)) == -1) {
        mylog(MYLOG_ERR, ...);
    }
    ...

    if (ret == 0) {
        ... /* may happen normally so don't bark : ) */
    }

    if ((ret=send(insock, buf, ret, 0)) == -1) {
        mylog(MYLOG_ERR, ...);
    }
    ...

    if (ret < strlen(buf)) {
        mylog(MYLOG_ERR, ...);
    }
}
} /* for() */
/* NOT REACHED */
return (0);
}

```

Errors are comprehensively logged and as a programming standard the *stacktrace* of the calling functions is always included.

Traffic goes unprohibited as long as there are no errors or attacks detected. Otherwise, filter modules supply the error message which the attacker is going to receive, and the real [MTA](#) remains unaware.

5.2.4 *modules.c*

The file provides functionality to loop through the registered (plugged in) modules until the buffer passes the security checks or at least one filter detects an intrusion attempt.

```

struct packet *
check_safety_modules(char *s)
{
    struct packet *pkt, pktr;
    int i, ret;

    pkt = (struct packet *) malloc(sizeof (struct packet));
    if (pkt == NULL) {
        mylog(MYLOG_ERR, ...);

        return (NULL);
    }

    memset(pkt, 0x0, sizeof (struct packet));
    pkt->status = STATUS_OK;
    strncpy(pkt->str, s, MAX_POSIX_LINE - 1);

    if (pthread_mutex_lock(&mt_reg) != 0)
        mylog(MYLOG_ERR,
            "check_safety_modules(): pthread_mutex_lock(): %s",
            strerror(errno));

    for (i=0;i<MAX_SOCKETS;i++) {
        if (sock_table[i] != 0) {
            memset(&pktr, 0x0, sizeof (struct packet));

            if ((ret=send(sock_table[i], pkt,
                sizeof (struct packet), 0)) == -1 ) {
                mylog(MYLOG_ERR, ...);
            }
        }
    }
}

```



```

pkt->status = STATUS_ERROR;
break;
}

if (ret < sizeof(struct packet)) {
    mylog(MYLOG_ERR, ...);
    ...
    break;
}

if ((ret=recv(sock_table[i], &pktr,
    sizeof (struct packet), 0)) == -1) {
    mylog(MYLOG_ERR, ...);
    ...
}

if (ret == 0) {
    mylog(MYLOG_ERR,
        "check_safety_modules(): module closed connection.");
    sock_table[i] = 0; /* clear out orphan socket */
    ...
}

if (pktr.status != STATUS_OK) {
    memset(pkt, 0x0, sizeof (struct packet));
    pkt->status = pktr.status;
    strncpy(pkt->str, pktr.str, MAX_POSIX_LINE - 1);

    break;
}

}

}

if (pthread_mutex_unlock(&mt_reg) != 0)
    mylog(MYLOG_ERR,
        "check_safety_modules(): pthread_mutex_unlock(): %s",
        strerror(errno));

return (pkt); /* all modules passed the security checks */
}

```

5.2.5 *mylog.c*

Very elegant file logging facility, styled around *syslog*³ format. It's very important to use files as a log storage and not specialized unix daemons because the whole process is going to run inside an opaque security sandbox.

```

void mylog(int priority, char *fmt, ...)
{
    int fl_firstcall = 1;
    char mname[][12]={ "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    va_list ap;
    struct tm *tm_ptr = NULL;
    time_t the_time;

    if (fl_firstcall) {
        fl_firstcall = 0;

        mylogfp = fopen(LOG_FILENAME, "a");
        if (mylogfp == NULL) {
            perror("fopen logfile");
            exit(EXIT_FAILURE);
        }
    }

    va_start(ap, fmt);

    time(&the_time);
    tm_ptr = localtime(&the_time);

    if (tm_ptr == NULL) {
        fprintf(stderr, "mylog(): localtime() fatal error\n");

        fprintf(mylogfp, "mylog(): localtime() fatal error\n");
        exit(EXIT_FAILURE);
    }

    /* logs to file, no syslog in a secure sandbox */

    fprintf(mylogfp, "%3s %02d %02d:%02d:%02d ",
        mname[tm_ptr->tm_mon], tm_ptr->tm_mday,
        tm_ptr->tm_hour, tm_ptr->tm_min, tm_ptr->tm_sec);

    fprintf(mylogfp, "%s: ", PROGNAM);

```

³ The default UNIX system logger.

```

fprintf(stderr, "%3s %02d %02d:%02d:%02d ",
    mname[tm_ptr->tm_mon], tm_ptr->tm_mday,
    tm_ptr->tm_hour, tm_ptr->tm_min, tm_ptr->tm_sec);

switch (priority) {
    case MYLOG_ERR:
        fprintf(stderr, "%s: ERROR ", PROGNAME);
        fprintf(mylogfp, "ERROR ");
        break;
    case MYLOG_WARNING:
        fprintf(stderr, "%s: WARNING ", PROGNAME);
        fprintf(mylogfp, "WARNING ");
        break;
    case MYLOG_INFO:
        fprintf(stderr, "%s: INFO ", PROGNAME);
        fprintf(mylogfp, "INFO ");
        break;
}

vfprintf(mylogfp, fmt, ap);
vfprintf(stderr, fmt, ap);
fprintf(mylogfp, "\n"); /* automagically add the new line */
fprintf(stderr, "\n");

    va_end(ap);
}

```

The *mylog()* function is a format string function, so it can take an unlimited number of arguments and format them accordingly. However that also means it is always important to provide it a format string argument, otherwise security errors can occur.

5.2.6 *packet.h*

The packet header simply defines the structure used for communication between the main proxy and it's plugged in filter modules:

```

#include "config.h"

#define STATUS_OK 0
#define STATUS_ERROR 1
#define STATUS_ATTACK 2

struct packet {
    char str[MAX_POSIX_LINE];
    int status;
};

```

10

5.2.7 *registry.c*

The registry is responsible for polling the socket directory and managing the *plug and go* technology. New filters can be attached during runtime without any sort of interruptions. The technology will be described later - as it is important for expandability.

```

int
_dirlist(char *path)
{
    ...
    fcount = scandir(path, &filelist, 0, alphasort);
    ...
    for(i = 0; i < fcount; i++) {
        ...
        if (stat(realpath, &sb) == -1) {
            ...
        }

        if (S_ISSOCK(sb.st_mode)) {
            if (_register(realpath) == -1) {
                mylog(MYLOG_ERR, ...);
            }
        }
        free(filelist[i]);
    }
    free(...);
    ...
}

```

10

20

```

int
_dirpoll(char *path) {
    ...
    for (;;) {
        if (stat(path, &sb) == -1) return (-1);
    }
}

```

```

        if (sb.st_mtime != old_mtime) {
            if (_dirlist(path) == -1) {
                mylog(...);
                ...
            }
        }
        old_mtime = sb.st_mtime;

        sleep(SLEEP_SEC);
    }
}

```

`_dirpoll()` detects changes in the directory structure, and `_dirlist()` scans the directory for unix domain sockets.

Dead sockets are automatically removed from the registry, and unconnected sockets get linked to the main proxy.

5.2.8 *rxp.c*

```

int
match(const char *string, char *pattern)
{
    int status;
    regex_t re;

    if ((status=regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB)) != 0) {
        mylog(MYLOG_ERR, ...);

        return(0);    /* report error */
    }

    status = regexec(&re, string, (size_t) 0, NULL, 0);
    regfree(&re);

    if (status != 0) {
        if (status == REG_NOMATCH) return (0);
        else mylog(MYLOG_ERR, ...);

        return(0);    /* report error */
    }

    return(1);    /* return OK */
}
.
.
.

```

```

int
compiled_match(const char *string, regex_t *code)
{
    int status;

    status = regexec(code, string, (size_t) 0, NULL, 0);
    if (status == REG_NOMATCH) return (0);
    if (status == 0) return (1);
}

```

30

The *compiled_match()* function is much faster than *match()* because it uses precompiled regular expression code. Therefore it is wiser to compile all the regular expressions at start-up and use the faster function call during runtime.

5.2.9 *subnet.c*

The *subnet* module contains utility functions for computing and verifying IPv4 addresses with their netmasks. I had to build my own [ASCII](#) to network byte order IP conversion function as the default routines do not handle bogus parameters correctly.

```

extern int /* boolean */
ip_in_subnet(uint32_t ip, struct s_subnet *snet)
{
    if ((ip << (32 - snet->mask)) ==
        (snet->ip << (32 - snet->mask))) return (1);

    else return (0);
}

extern uint32_t
ip_to_binary(char *ip1)
{
    uint32_t b1,b2,b3,b4;
    uint32_t result;
    char *ptr=NULL, *old=NULL;
    char ip[16]; /* internal buffer to store and work with the IP */

    /*
     * I prefer the use of malloc here internally because it's faster
     * than a call to strlen. However strncpy might be better.
     */
}

```

10

20

```

memset(ip, 0x0, 16);
strcpy(ip, ip1, 15); /* last byte must be zero */

b1=0;b2=0;b3=0;b4=0;

ptr = strchr(ip, '.');
if (ptr == NULL) {
    mylog(MYLOG_ERR, "ip_to_binary(): invalid IP address");
    return (0);
}
*ptr='\0';
b1=atoi(ip);
ptr+=sizeof(char);
old=ptr;

ptr = strchr(ptr, '.');
if (ptr == NULL) {
    mylog(MYLOG_ERR, "ip_to_binary(): invalid IP address");
    return (0);
}
*ptr='\0';
b2=atoi(old);
ptr+=sizeof(char);
old=ptr;

ptr = strchr(ptr, '.');
if (ptr == NULL) {
    mylog(MYLOG_ERR, "ip_to_binary(): invalid IP address");
    return (0);
}
*ptr='\0';
b3 = atoi(old);
ptr+=sizeof(char);

b4=atoi(ptr);

result=0;

/* 3,2,1 use this for host byte-order instead of what follows */

b2 = b2 << 8;
b3 = b3 << 16;
b4 = b4 << 24;

result = b4 + b3 + b2 + b1;

/* returns in network byte order */
return (result);
}

```

5.3 SECURITY

The daemon builds a sandbox before executing any other functionality. This assures that the worst case scenario, when the daemon itself is taken over by an intruder, is jailed into a security sandbox.

The security sandbox implemented by the daemon consists of a restricted filesystem zone, with the least possible system privileges. That is, the daemon changes the root path to a configurable "no-man's-land" area, and then it irreversibly drops all its execution privileges to the lowest level possible. This is done by the *security_drop_privileges()* function from the *security.c* module:

```
int
security_drop_privileges(void)
{
    sandbox_uid=_get_sandbox_user();
    sandbox_gid=_get_sandbox_group();

    if (sandbox_uid == -1) {
        mylog(MYLOG_ERR, ...);

        return (-1);
    }

    if (sandbox_gid == -1) {
        mylog(MYLOG_ERR, ...);

        return (-1);
    }

    if ((sandbox_uid == 0) || (sandbox_gid == 0)) {
        ... /* don't be dumb root */
    }

    if (chdir(sandbox_path) == -1) {
        ... /* change directory */
    }

    if (chroot(sandbox_path) == -1) {
        ... /* change root path */
    }
}
```



```

30
    if (setgid(sandbox_gid) == -1) {
        ... /* drop group privileges */
    }

    if (setuid(sandbox_uid) == -1) {
        ... /* drop user privileges */
    }

    return (0);
}
40

```

It is *very important* to observe that each filter module may run at different privilege levels and have a sandbox of its own. This arcane configuration makes life a cruel scenario for possible intruders.

An unfortunate attacker which manages to drill into the system may need to sequentially break out of 50⁴ or more unprivileged sandboxes. And it may need to use different techniques for each of them.

The sources are compiled with the *gcc -Wall* flag, which provides some basic static analysis for possible bugs. No errors are mentioned.

Furthermore, all the secure programming principles briefly introduced in the previous chapters are carefully respected.

5.4 EXPANDABILITY

The *Plug-and-Go* technology allows for uninterruptible addons and updates to be attached. The mail traffic flows unaffected through the security mechanisms while an unlimited number of filter modules can be attached to the system.

⁴ The number of registered modules.

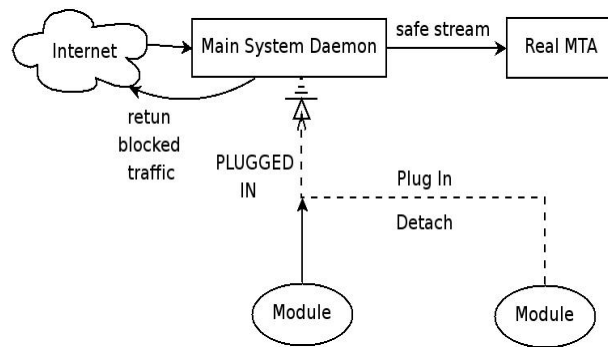


Figure 5. Plug and Go Technology - unlimited filter modules

5.5 ERGONOMY

At last, ergonomics is pretty self-implied. The proxy itself requires no restart when it is updated or when modules detach, so no administration is required. It can remain untouched for a long period of time, while filter modules may come and go.

Administration burden is thus thrown upon the filtering modules. However modules may be written by different developers and companies, so each can handle the technical support in private.

It is also possible to grant different levels of administration and responsibilities accordingly to the number of filtering modules, or simply share the burden with a low-privileged sysadmin.

5.6 SAMPLE FILTER MODULES

5.6.1 SMTP Syntax Verifier

The regular expressions are precompiled at startup for speed optimization. The following expressions were long tested on production mail servers and are quite powerful:

```

#define RXP_DOMAIN "([[:alnum:]]+)|((([[:alnum:]]+)|([[:alnum:]]+)"
"\ \-[:alnum:]]+)\.\.)+[[:alpha:]]+)"

#define RXP_ADDRESS "([[:alnum:]]+(@ RXP_DOMAIN "))"
#define RXP_BRACKET_ADDRESS "(" RXP_ADDRESS ")|(<" RXP_ADDRESS ">)"
#define RXP_SMTP_ADDRESS "<" RXP_ADDRESS ">"
#define RXP_IP_BYTE "[0-9]?[0-9]?[0-9]"

#define RXP_IP RXP_IP_BYTE "." RXP_IP_BYTE "."
RXP_IP_BYTE "." RXP_IP_BYTE /* very round */

/*
 * Commands are detected in the first round, profound
 * syntax is checked in the second round.
 *
 * Here follow primitive command detectors for round 1.
 */

regcomp(&RXP_CMD_HELO, "^(^Hh][Ee][Ll][Oo][ ])|(^Hh][Ee][Ll][Oo])$", ...); 20
regcomp(&RXP_CMD_EHLO, "^(^Ee][Hh][Ll][Oo][ ])|(^Ee][Hh][Ll][Oo])$", ...);
regcomp(&RXP_CMD_VRFY, "^(^Vv][Rr][Ff][Yy][ ])|(^Vv][Rr][Ff][Yy])$", ...);
regcomp(&RXP_CMD_EXPN, "^(^Ee][Xx][Pp][Nn][ ])|(^Ee][Xx][Pp][Nn])$", ...);
regcomp(&RXP_CMD_HELP, "^(^Hh][Ee][Ll][Pp][ ])|(^Hh][Ee][Ll][Pp])$", ...);
regcomp(&RXP_CMD_QUIT, "^(^Qq][Uu][Ii][Tt][ ])|(^Qq][Uu][Ii][Tt])$", ...); 30
regcomp(&RXP_CMD_RSET, "^(^Rr][Ss][Ee][Tt][ ])|(^Rr][Ss][Ee][Tt])$", ...);
regcomp(&RXP_CMD_NOOP, "^(^Nn][Oo][Oo][Pp][ ])|(^Nn][Oo][Oo][Pp])$", ...);
regcomp(&RXP_CMD_XFORWARD,
"^(^Xx][Ff][Oo][Rr][Ww][Aa][Rr][Dd][ ])|(^Xx][Ff][Oo][Rr][Ww][Aa][Rr][Dd])$", ...);
regcomp(&RXP_CMD_MAIL, "^(^Mm][Aa][Ii][Ll][ ])|(^Mm][Aa][Ii][Ll])$", ...);
regcomp(&RXP_CMD_RCPT, "^(^Rr][Cc][Pp][Tt][ ])|(^Rr][Cc][Pp][Tt])$", ...); 40
regcomp(&RXP_CMD_DATA, "^(^Dd][Aa][Tt][Aa][ ])|(^Dd][Aa][Tt][Aa])$", ...);

/*
 * Now for the Real syntax checkers,
 * to be used in round 2.
 */

```

```

regcomp(&RXP_CMD_HELO_OK, "^[Hh][Ee][Ll][Oo][ ]" RXP_DOMAIN "$", ...);
regcomp(&RXP_CMD_EHLO_OK, "^[Ee][Hh][Ll][Oo][ ]" RXP_DOMAIN "$", ...);
regcomp(&RXP_CMD_VRFY_OK, "^[Vv][Rr][Ff][Yy][ ]" RXP_BRACKET_ADDRESS "$", ...);
regcomp(&RXP_CMD_EXPN_OK, "^[Ee][Xx][Pp][Nn][ ]" RXP_BRACKET_ADDRESS "$", ...);
regcomp(&RXP_CMD_HELP_OK, "^[Hh][Ee][Ll][Pp]([ ][:alpha:]+)?$", ...); 60
regcomp(&RXP_CMD_QUIT_OK, "^[Qq][Uu][Ii][Tt]$", ...);
regcomp(&RXP_CMD_RSET_OK, "^[Rr][Ss][Ee][Tt]$", ...);
regcomp(&RXP_CMD_NOOP_OK, "^[Nn][Oo][Oo][Pp]$", ...);
regcomp(&RXP_CMD_DATA_OK, "^[Dd][Aa][Tt][Aa]$", ...);
regcomp(&RXP_CMD_XFORWARD_OK,
        "^[Xx][Ff][Oo][Rr][Ww][Aa][Rr][Dd][ ]" RXP_IP "$", ...); 70
regcomp(&RXP_CMD_MAIL_OK,
        "^[Mm][Aa][Ii][Ll][ ]" RXP_SMTP_ADDRESS "$", ...);
regcomp(&RXP_CMD_RCPT_OK,
        "^[Rr][Cc][Pp][Tt][ ]" RXP_SMTP_ADDRESS "$", ...);

```

There are two major rounds for syntax checking: first, commands are identified even with bogus syntax. This allows the module to send appropriate error messages back to the Internet user - [SMTP](#) has different error codes for unknown commands and bad parameter syntax.

The second round will check the buffer against the appropriate precomputed regular expression, depending on what command was identified (and only if the first round is passed). As expected, `comp_match()` is used.

5.6.2 Realtime Shellcode Disassembler

The disassembly routines are imported from the *Libdasm*⁵ library. It is fairly easy to use, but interpreting the result can be very frustrating:

⁵ <http://www.nologin.net>

```

..
#include "../libdasm.h"
..

int main() {
    INSTRUCTION inst; /* internal structure */
    ...

    /*
     *      Fetches 32-bit Intel instruction from buffer &data      10
     */

    get_instruction(&inst, data, MODE_32);

    if (inst.type == INSTRUCTION_TYPE_ADD) {
        ...

        if (inst.op1.type == OPERAND_TYPE_MEMORY) {
            ...
            if (inst.op1.basereg == REGISTER_EDX) ...      20
            if (inst.op2.reg == REGISTER_EAX) {
                ...
                if (inst.op2.type != OPERAND_TYPE_REGISTER) {
                    ...
                }
            }
        }

        return (0);
    }
}

```

30

Unfortunately the x86 instruction set is very dense, and almost any regular human language sentence can be translated into correct assembly instructions by linear disassembly.

For example: "Thank you, Hannah, Princess of Persia" can be disassembled into:

```

;      Intel Syntax
;
push esp
push dword 0x206b6e61
jns 0x77
jnz 0x2a
dec esp
popa
jnz 0x80
popa

```

10

```

sub al,0x20
push byte 0x75
outsb
imul ebp,[ecx+0x65],0x20
jz 0x91
db 0x69
outsb
db 0x6b
insb
db 0x65
db 0x72

```

20

Therefore simple disassembly may suffice on other hardware architectures, but not on x86. And the following shellcode makes things even more perverse:

```

;      AT&T Syntax
;
;Dump of assembler code for function beast:
0x08049780 <beast+0>: jmp 0x8049791 <beast+17>
0x08049782 <beast+2>: pop %esi
0x08049783 <beast+3>: sub %ecx,%ecx
0x08049785 <beast+5>: mov %esi,%ebx
0x08049787 <beast+7>: mov $0x17,%cl
0x08049789 <beast+9>: addb $0xaa,(%ebx)
0x0804978c <beast+12>: inc %ebx
0x0804978d <beast+13>: loopne 0x8049789 <beast+9>
0x0804978f <beast+15>: jmp 0x8049796 <beast+22>
0x08049791 <beast+17>: call 0x8049782 <beast+2>
0x08049796 <beast+22>: xchg %edx,(%esi)
0x08049798 <beast+24>: cmpsb %es:(%edi),%ds:(%esi)
0x08049799 <beast+25>: mov $0xbec98585,%esi
0x0804979e <beast+30>: mov $0xc4bfb885,%esi
0x080497a3 <beast+35>: fistpll (%ecx)
0x080497a5 <beast+37>: cmpsb %es:(%edi),%ds:(%esi)
0x080497a6 <beast+38>: stos %al,%es:(%edi)
0x080497a7 <beast+39>: test $0x239106a6,%eax
0x080497ac <beast+44>: (bad)
0x080497ad <beast+45>: add %al,(%eax)
0x080497af <beast+47>: add %al,(%eax)
0x080497b1 <beast+49>: add %al,(%eax)
0x080497b3 <beast+51>: add %al,(%eax)
0x080497b5 <beast+53>: add %al,(%eax)
0x080497b7 <beast+55>: add %al,(%eax)
0x080497b9 <beast+57>: add %al,(%eax)
0x080497bb <beast+59>: add %al,(%eax)
0x080497bd <beast+61>: add %al,(%eax)
0x080497bf <beast+63>: add %ch,%bl
;End of assembler dump.

```

10

20

30

And in buffer stream it looks like:

```
char beast[]=  
"\xeb\x0f\x5e\x29\xc9\x89\xf3\xb1\x17\x80"  
"\x03\xaa\x43\xe0\xfa\xeb\x05\xe8\xec\xff\xff\xff"  
"\x87\x16\xa6\xbe\x85\x85\xc9\xbe\xbe\x85\xb8\xbf"  
"\xc4\xdf\x39\xa6\xaa\xa9\xa6\x06\x91\x23\xd6";
```

The sample shellcode is a *touppercase()* *evasion* instruction set which runs on FreeBSD⁶. The bold instructions decrypt the rest of the code by adding *0xAA* in hexadecimal to the value of every byte that follows.

A lot of imbricated conditional blocks need to be used for simulating my *proof of concept* decryption instructions.

After the decryption loop has finished, the remaining buffer has been transformed into a pure *execve(/bin/sh)* system call. If executed, provides terminal control:

Intel Syntax	AT&T Syntax
;	
xor eax,eax	<beast+22>: xor %eax,%eax
push eax	<beast+24>: push %eax
push dword 0x68732f2f	<beast+25>: push \$0x68732f2f ; load /bin/sh
push dword 0x6e69622f	<beast+30>: push \$0x6e69622f ;
mov ebx, esp	<beast+35>: mov %esp,%ebx
push eax	<beast+37>: push %eax
push esp	<beast+38>: push %esp
push ebx	<beast+39>: push %ebx 10
push eax	<beast+40>: push %eax
mov al, 59	<beast+41>: mov \$0x3b,%al ; execve()
int 0x80	<beast+43>: int \$0x80 ; syscall int

The shellcode presented above has two interesting properties:

- it remains functional even if passed to an uppercase conversion function
- it hides the system call (interrupt 0x80) by encrypting the machine binary code

Uppercase/lowercase evasive instruction sets are useful when the target application converts the supplied user

⁶ <http://www.freebsd.org/>

input *before* the vulnerability is triggered. SQL databases are notorious examples.

The only precise methodology for detecting such perverse situations is by trying to execute the decryption routines in a *virtual processor emulator*. In my simplified scenario, all I have to do is simulate the decryption loop with an arithmetic instruction.

An interesting opensource project named *Libemu*⁷ promises to emulate all the major hardware architectures, at least the simplified instruction set required for analyzing shellcode decryptors. It is still under research and development phase at the date of this writing.

Shellcodes must be short and are slaughtered by a lot of restrictions in order to work in the Wild. However emulating a dense instruction set like x86 is very hard and out of this paper's scope.

For measurements, a simple decryption loop based on the disassembled code will suffice. Nothing useful can be done without kernel system calls. If such a syscall is detected during decryption or before, the code is most certainly dangerous.

So to speak, the module attempts to identify (through emulation) dangerous payloads inside the buffered stream. It stops when a syscall routine is detected during emulation or the tested buffer segment becomes shorter than 3 bytes.

A valid unix syscall needs at least 3 bytes to be encoded, and shellcodes must use system calls:

⁷ <http://libemu.tigris.org/>


```

#define CRYPTO_START 22
#define KEY 0xAA

...

int
emulate_decrypt(char *buf)
{
    int i;

    for (i=CRYPTO_START; i<strlen(buf); i++) *(buf+i*sizeof(char))10
        = *(buf+i*sizeof(char)) + KEY;

    return (i);
}

...

/* boolean */
int
evil_code(char *realdata)
{
    INSTRUCTION inst;
    int i, c, bytes, format = FORMAT_INTEL, size, len;
    char *data;

    data = realdata;

    while (strlen(data) >= 3) {
        /* A system call takes at least 3 bytes ;)20
         * Example 0xCD 0x80 -> kernel mode interrupt
         * EAX must hold the syscall number so +1 byte ;)
         */

        c = 0;
        size = strlen(data);

        /*
         * Intentionally waste a function call to
         * simulate the frame stack creation before
         * the decryption loop :-)40
         */
        bytes=emulate_decrypt(data);

        while (c < size) {
            len = get_instruction(&inst, data + c, MODE_32);

            if (!len || (len + c > size)) {
                return (-1); /* bad opcode or out of bound */
            }
        }
    }
}

```

```

    c += len;

    if (inst.type == INSTRUCTION_TYPE_INT) {
        if (inst.op1.type == OPERAND_TYPE_IMMEDIATE) {
            if (inst.op1.immediate == 0x80) {
                /* bastard system call */

                return (1); /* evil code */
            }
        }
    }
    data = data + sizeof(char);
} /* try to emulate as long as shellcode is longer than 3 bytes */

return (0); /* code is clean */
}

```

60

EVALUATION

6.1 BENCHMARKING

They who would give up an essential liberty for temporary security, deserve neither liberty or security.
 — Benjamin Franklin

Postal¹ was used for stress testing the solution, allowing variable message sizes from 1 Kb to 1 Mb and a maximum of 4 messages per TCP link. This fits the real world scenario well.

time(s)	messages	Kbytes	errors
60	25	12602	0
120	25	12565	0
180	25	12633	0
240	23	12599	0
300	23	11127	0
360	25	12470	0
420	28	12531	0
480	25	12620	0
540	24	12587	0
600	24	12491	0

Table 3. Benchmark with proxy

The average traffic flow per minute is about 12.14 Megabytes. That's a relief! Realtime disassembly and emulation is an $O(n)[1]$ algorithm for each and every SMTP command - it's probably the slowest method of detecting shellcodes, but it compensates with precision. Such glorious attempts would

¹ <http://www.coker.com.au/postal/>

never be realistic for web and other time-critical services, at least with the current technological background.

The daily barrier of 17.07 Gigabytes is a little bit better than a Tier-1² network, with 1.61 Mbps bottleneck.

Zero errors for about 121.40 Megabytes of mail traffic prove that the proxy is very robust. And as *stability* is a metric of *security*, the main objectives have been accomplished.

time(s)	messages	Kbytes
60	277	143047
120	295	140516
180	289	142715
240	284	146482
300	296	139305
360	285	145620
420	283	146968
480	291	142404
540	289	143185
600	287	143718

Table 4. Benchmark without proxy

All tests were done using the Postfix³ [MTA](http://www.postfix.org/). It's obvious that without the proxy the maximum traffic flow can blow up to 196.91 Gigabytes daily. That is the equivalent of 140 Megabytes per minute. So security really does cost!

The proxy with code emulation and profound syntax integrity slows down the "mail pipe" capacity with about 83%. In *Figure 7* it can be observed that the performance penalty is relatively constant around a linearly stable value.

² 1.54 Mbps

³ <http://www.postfix.org/>

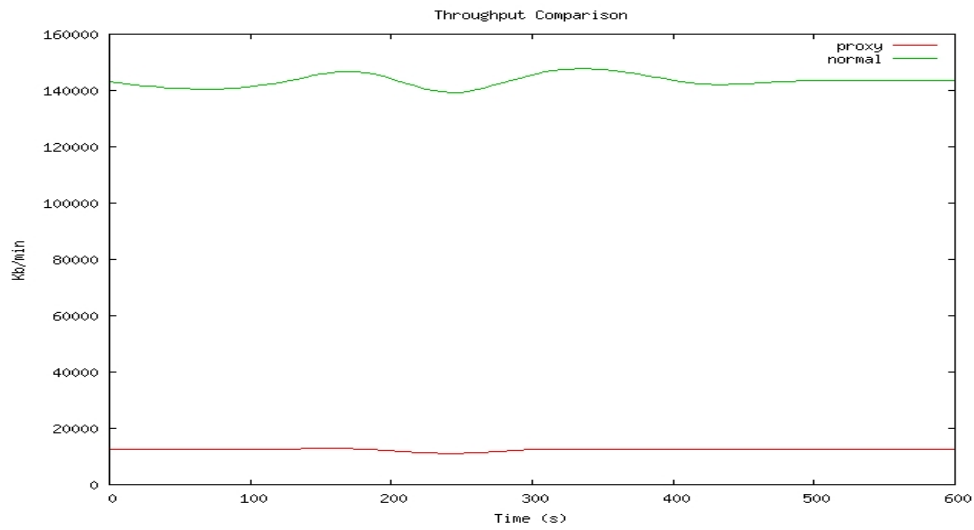


Figure 6. Throughput comparison

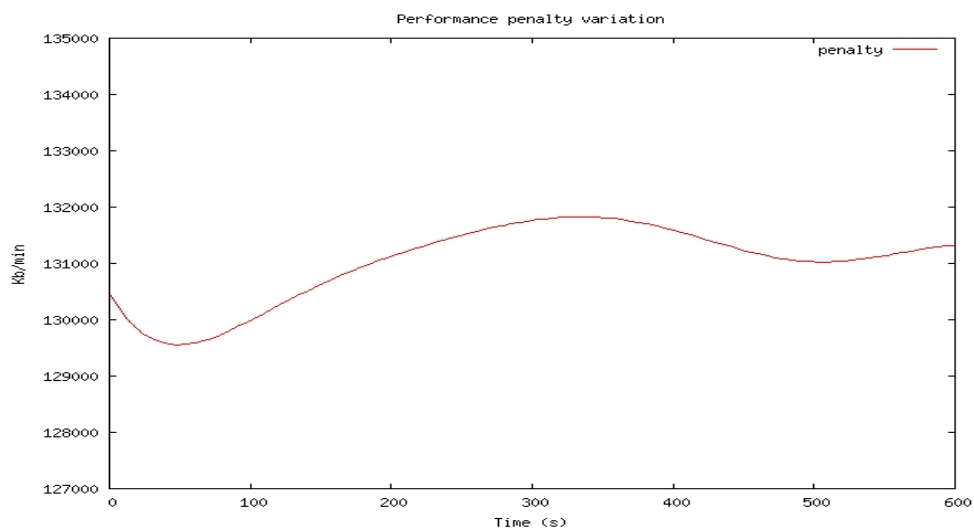


Figure 7. Performance penalty variation

6.2 SYSTEM REQUIREMENTS

The tests were done on a dual-processor P4 system with *peer-to-peer* inter-processor synchronization.

The minimum requirements are very low, the only needed resource is CPU cycles. A medium configuration may consist of a Pentium-3 with 1 Ghz clock.

It's requirements are tightly bounded around the needed mail throughput. No significant memory or harddisk resources are used.

In short, it may even run well on a 133 Mhz processor like Postfix does, but only with limited throughput - small number of users.

6.3 SOURCE CODE COMPLEXITY

The *proof of concept* solution has no more than 5k lines of security code. As debated earlier, the security programming paradigm requires less code for the same functionality, to keep it controllable and clear.

metric	value
McCabe's Cyclomatic Number	250
Lines of code / comments	9.75
Cyclomatic complexity / comments	2.84

Table 5. Proxy complexity metrics

metric	value
McCabe's Cyclomatic Number	159
Lines of code / comments	10.698
Cyclomatic complexity / comments	3.698

Table 6. Syntax module complexity metrics

The CCCC⁴ tool was handy for generating source code metrics.

metric	value
McCabe's Cyclomatic Number	82
Lines of code / comments	6.346
Cyclomatic complexity / comments	1.577

Table 7. Disassembler module complexity metrics

⁴ <http://cccc.sourceforge.net/>

CONCLUSION

In spite of the fact that it degrades performance by 83%, the solution compensates with the highest level of security. There is no technology more precise on identifying intrusion attacks than *virtual processor emulators*. Emulators can detect unknown polymorphic and alphanumeric instruction sets at first contact.

My solution demonstrates that the oldest Internet service can also be the most secure, due to its *store and forward* nature.

In fact, if 17.07 Gigabytes of mail traffic is enough for the daily use, the solution fits perfectly for highly demanding protection levels.

Medium-sized datacenters use Tier-1 connections and share them across hundreds of web servers. The mail system requires far less bandwidth than web and realtime services and the current unoptimized solution can handle a bit more.

7.1 FUTURE WORK

A lot of work can be done for optimization. The emulator should be fully developed for solving complex encryption routines.

The proxy could manage a pool of persistent connections with the real [MTA](#) to minimize connection time gaps.

There are other types of threats which could be added to the protection scheme, in order to develop a complete, integrated solution.

New modules could be added for inspecting [SMTP](#) commands.

BIBLIOGRAPHY

- [1] Vladimir-Ioan Crețu. *Structuri de Date și Algoritmi*. Editura Orizonturi Universitare Timișoara, 1st edition, 2000. (Cited on page 61.)
- [2] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. June 2000. (Cited on page 6.)
- [3] Oded Horovitz. Big Loop Integer Protection. *Phrack*, 11 (60), December 2002. (Cited on page 8.)
- [4] Ionel Jian. Assembly Programming Course. 2005. (Cited on page 6.)
- [5] Elias Levy. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 1996. (Cited on pages 5 and 25.)
- [6] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. *11th Conference on Computer and Communications Security*, pages 298–307, October 2004. (Cited on page 7.)
- [7] Mircea Vladutiu. Computer Architecture Course. 2005. (Cited on pages 9 and 25.)
- [8] Perry Wagle and Crispin Cowan. StackGuard: Simple Stack Smash Protection for GCC. *GCC Developers Summit*, May 2003. (Cited on page 5.)
- [9] David A. Wheeler. *Secure Programming for Linux and Unix HOWTO*. <http://www.tldp.org/>, 3rd edition, 2003. (Cited on page 33.)

COLOPHON

This thesis was typeset with $\text{\LaTeX} 2_\epsilon$ using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

The typographic style was inspired by Robert Bringhurst's genius as presented in *The Elements of Typographic Style* (Version 2.5, Hartley & Marks, 2002) and is available for \LaTeX via CTAN as "[classicthesis](#)".

DECLARATION

Timișoara, Romania, Europe, June 2007

Alin-Adrian Anton