Preserving Cybercrime Evidence

Martín de la Herrán Brickmanne

September the 16th, 2003

Foreword

dédié à Papa Henry

First of all, I would like to thank my parents and *Mazet* for all their love and unconditional support.

My gratitude to the people at DTU, especially Robin Sharp, for trusting in me while still in Spain, having only shared a couple of e-mails and phone calls with me. This great international experience would not have been possible without you.

Greetings to all my friends in Spain and abroad, you know who you are.

And finally thanks to all the other people who have helped me make this possible. You are many, and although not mentioned you here, you are not forgotten.

Martín de la Herrán Brickmanne.

ABSTRACT

Cybercriminals who are trying to hack into a system usually take precautions to remove or hide as many traces of their activity as possible, for example by deleting (parts of) log files, replacing certain system functions by special "hacker versions" which if activated will not reveal the presence of the hacker, and so on. This can make it difficult for a prosecutor to secure reliable evidence of what has happened, in case it is necessary to proceed with criminal charges.

In this project, techniques for ensuring that reliable evidence can be preserved are to be investigated. These will include secure logging, secure system monitoring, and hardening of the system against changes introduced by authorised or unauthorised users. The analysis should consider as many aspects of these techniques as possible, including for example:

- The type of evidence which they can secure and its significance for the investigation of cybercrimes;
- The technical requirements for their implementation;
- The extent to which they degrade system performance.

Based on this analysis, a design proposal for a system which is resistant to the destruction of cybercrime evidence is to be produced, and (to the the extent that time permits) a demonstration model of such a system is to be implemented.

Contents

CHAPTER 1

Introduction

1.1 Introduction.

One of the main problems about computers which are attacked and broken into is the difficulty of tracing back the attack. A smart attacker can easily hide his actions once he has gained superuser[1] access to the system. This implies that if we want to trace back the actions which have happened on untrusted machines after they have been broken into we must set up special mechanisms which should prevent undetectable modification of the system logs.

The basic measure to record the information of what is happening in a specific machine is the use of *loggers*. A logger is a software program used by other programs to store information about what they are doing. Using a centralized logger make the management of the logs easier, and virtually all the operating systems have one, or several, loggers available[2]. So, excepting specific programs which have integrated loggers

[1] This "superuser" is normally called "root" in *nix, or "Administrator" in Windows machines. Through this work, we will indifferently use this term to refer to users with unlimited software restrictions on a machine.

[2] i.e.: Under Microsoft operating systems, it is called "event viewer". There are several under the different unixes, like syslog, syslog-ng, metalog...

(but can also use external loggers), such as the Apache web server, securing the system logger and the logs it creates should be enough to trace back all the relevant facts which have happened in a specific computer. This approach is valid and has been used extensively since the beggining of computers to keep a centralized depository of system events. But after system is broken into, a malicious superuser can modify all logs following his attack, and also possibly previous logs. This creates a serious problem:

- The attack may remain undetected for a long period of time.
- The modifications performed on the system data after the attack are unknown. The gravity of the implications will vary according with the relation the data has with the real world, and may be very serious.
- Once detected, the system administrators may find themselves unable to trace back the person originating the attack, the date the attack was successful, or the method used to infiltrate in the machine.
- The attacked system may be used at the attacker's will, like as a platform for new attacks.

In a worse-case scenario, an attacker will have complete control over the machine. If this is the case, fully preventing the modification of future logs may be impossible to accomplish. Albeit he would need to be skilled, with full control any security measure executed after the breach could theoretically be circumvented by the attacker.

This is the situation: we know that the system and its data were correct until the attack. After that the system may not be fully trusted anymore. What solutions does a system administrator have to keep his system, and his data, as safe as possible? We will explore the different possibilities along the following chapters:

First, we will review software approaches: why attacks are still successfull albeit software companies attempts to make software secure, and different safety measures which may be deployed on servers. Then, we will explore the system logging subsystem: correct configuration and typical measures used nowadays to keep logs safe. We will then concentrate on a still rarely used possibility: adding encryption and authentication on the system logs. We will finish with an overview of possible hardware devices which may help keeping data safe.

1.2 Cryptography in logging: Cryptohasher

We have developped a working example of using cryptography together with the system logger. We will proof that preventing undetected modification of past logs is possible, once the system has been set up correctly. We may not be able to tell *what* was deleted, but we will see that setting up tamper-proof logs is feasible. As modification of the log entries is detectable, we will be able to certify the data recorded is genuine, and use such data to trace the attacker actions, or use it as an evidence in case it is necessary to proceed with criminal charges. On the other hand, if the attacker modifies or deletes the logs, it would quickly be evident that something wrong is happening[3] in the system and more specific measures could be taken to put down the system and analyse what is happening. Thus, if a system is correctly configured, we will demonstrate measures to make an attacker undetected modification of logs previous to his breach impossible.

We will also discuss approaches to make those past logs undeletable, using some of the system hardening tools previously introduced. At the same time we will try to make modification of future logs as difficult as possible, albeit post-attack integrity cannot be formally demonstrated.

1.3 Objectives

The objectives we wish to attain at the end of this thesis will be an analysis of the tools available, and possible improvements, of the following desired features of a secure system:

- System configuration:

[3]as long as we set up a periodic log checking, and trigger an alarm as soon as modification or strange behaviour are detected. We will deepen on this subject later.

  - Harden the system to offer a difficult or impossible challenge to attackers.
  - Use periodic sytem checking and alarm generation tools to inform system administrators about possible breaches.

- Make system logs previous of system intrusion tamper-proof

  - non-legitimate superusers will not be able to hide their tracks undetected.
  - log integrity previous to the attack is demonstrable, and useable as a proof.
  - modification of logs indicates system intrusion or important malfunction.

- Make system logs undeletable

  - Explore possibilities making non-legitimate superusers unable to delete, wipe or modify past logs.
  - for hardware solutions, past logs have to be completely undeletable.

We will concentrate on the Linux system, as it is free, mature and widely adopted by many individuals and organizations. Sharing many similarities with Linux, the results of this work may easily be ported *BSD[4] operating systems, or similar unix-like environments.

[4] Various free BSD operating systems include FreeBSD `http://www.freebsd.org`, NetBSD `http://www.netbsd.org` and OpenBSD `http://www.openbsd.org`.

Chapter 2

System Lockdown

Nowadays, any machine connected to the Internet may find itself victim of an attack. Computers offering services to the world are specially vulnerable, as they are very visible and accessible from the whole Internet. But computers inside private or firewalled networks may also find themselves attacked, as any compromised computer within such a private network can be used to propagate the attack. This has happened very often as a consequence of recent viruses, which have been found inside networks that should be completely closed[1], infected by some laptop connected previously to the Internet, or via some computer infected via e-mail. Also, relying on obscurity is not safe: computers freshly connected to the Internet using a previously unused IP address have been found to be scanned for vulnerabilities very quickly, in a matter of hours.

This unfortunate scenario implies that hardening a computer system is very important, even more if the computer is a server. The more important the data served or used by our machine, the more we want to be sure of the integrity and private state of it. As it has been proved with recent credit card number stealing and frauds over the Internet, the implications of being hacked can get very expensive.

[1]like it happened at the Ohio Nuclear Plant network, where the Slammer worm disabled a safety monitoring system.

In this chapter, we are going to review the security model used under the Linux operating system, and discuss tactics and methods which should be used by system administrators to improve the security of their servers. While discussing the flaws of the current Linux security model, we will recommend some secure system set-ups, before reviewing more sophisticated techniques which imply deeper system modification.

2.1 Linux security model

Under standard Linux, there are only two types of users: normal and superuser. The former has many limitations and is used by normal people or processes in the machine. But while the later is used by system administrators, it is also used by many programs which need some kind of special privileges, and thus cannot be run in normal user space. But if these programs have bugs, and can be exploited to run random code, the code will run as superuser, with no restrictions at all. This has been the main source of successful technical[2] computer attacks, to find a process on a machine which is run as root, find a vulnerability in it, and manage to exploit that program and make it run whatever code we want: we have full control over the machine. The first step is easy for attackers, they only need to scan the machine remotely for network services, or locally if they have an account in it. The second, finding vulnerabilities, has turned out to be easier and easier today thanks to the Internet and the full disclosure of vulnerability information, which spreads much faster than the speed of program patch deployment. The final step, actually exploiting the bug, is probably the hardest one to perform technically speaking, but again the Internet greatly helps and connects people willing to develop and share such information. This finally gives many unskilled people access to all the tools and information they need to attack computers.

As security is only as strong as the weakest link, we need to limit the power of the programs being run, so that they only get as many privileges as they need, and can only perform the task they were designed for. We also need to set up tools to detect the attacks and, if possible, stop

[2] as opposed to *social engineering*, in which an attacker obtains information like passwords impersonating people, or password guessing or cracking.

them or limit the damage they can do. No program should have full unrestricted control over the machine, we must weaken the root.

2.2 Weakening the root

Once an attacker has entered the system and gained full root privileges, the system itself cannot be trusted. Full system control allows hiding of the intrusion as it can prevent the legitimate system tools to detect the presence of the attacker files and processes inside. In this section we will compare different measures a system administrator can set up to limit the impact and the damage a possible intruder is able to commit.

To limit the actions of an external intruder, it is necessary to limit the actions the normal *root* can do. These limits have to be applied to both legitimate and unauthorized superusers, as the operating system cannot distinguish one from the other once it has been fully compromised[3]. Thus, we have two objectives to accomplish in this area:

- Limit the range of actions a non-legitimate superuser can do.
- Keep the system manageable by legitimate system administrators.

These two objectives are difficult to satisfy completely at the same time; the more we harden a system, the more there will be impediments to an easy administration. But many actions which are normally not needed to properly administrate a system can be locked, actions which would ease the work of an intruder. These kind of actions include the following, which are presented approximately in an easier-to-harder order.

- Only install and run the necessary programs.
  - Many systems are badly configured, as they run too many services which are not really needed. Keeping the running services at a minimum ensures we are not broken into due to some service we did not really need. Thus, ftp daemons, mail

[3]after all, that is what makes a successful attack - if unauthorized superusers could be easy detected, they would as easily be defeated!

servers, RPC daemons, bind, telnet, NFS, and any other network services are better switched off when not needed. Also, any unnecessary local program which may obtain privileges should be disabled because an attacker obtaining access to a local account would try to use those programs to gain access to the root level.

- Properly configure the running servers.

  - System services can be well programmed and perhaps bug-free, but a misconfigured program can easily be intentionally misused, allowing unwanted actions. This is the case of ftp servers, which should force at least quotas and not allow anonymous uploads[4]. Mail servers should check they cannot be used to send mail to the Internet from untrusted sources; failing to do so can easily degenerate in the server being used for unsolicited bulk e-mail (*spam*) distribution. RPC daemons and database servers should only allow access from selected hosts, and if possible using some authentication protocol. DNS servers (*bind*) should only allow zone transfers from master to slave servers. The web server should not reveal the content of directories if it is not intended to. We could continue giving many examples, every program being used as a server for other hosts should have its configuration carefully examined to only allow the actions we want it to perform, and limit the amount of private information we give to the outside.

    A good measure for servers is to *chroot* the programs. A chroot-ed program only lives inside a particular directory, which acts as his root filesystem. Thus, it cannot see, use, nor modify any file outside his local filesystem tree. This can limit the amount of damage that a program would cause in

[4]or, if anonymous uploads are required, hide them once uploaded, so that the machines cannot be used as storage for illegal programs or files. A very common usage for misconfigured or broken ftp servers is, precisely, the distribution of copyrighted programs, music or movies.

case of malfunction or intrusion, but is not completely fail-safe because a root user within a chroot cage can still perform many tasks. The main problem of chroot cages are that they make the programs which reside inside more difficult to set-up and manage.

- Disable loading kernel modules.

  – Kernel modules can modify heavily the behaviour of a system, effectively circumventing most of the available security measures. Thus, the ability to load kernel modules has to be limited, or completely denied, unless we can be confident the superuser is really who he says to be. The flexibility of loading modules can be very dangerous because loadable kernel modules are the main component of an attacker's *root kit*. These kernel modules have several functions, they can hide running processes, files, network connections, and even hide themselves[5]. Thus, a machine infected with this kind of tools locally appears to be working normally, no extra activity is recorded by the system, while the truth is very different. As the kernel module can even show files in their previous state before being modified, nothing less than a full cold system reset from a safe-known kernel may be necessary to detect the intrusion. Machines attacked this way are often used to record other users'successful passwords or information, as gateways for further hacking into the network the machine is located, or any other potentially malicious activity. Often, the only indication of a successful attack of this kind will be some strange network traffic coming from or to the exploited machine.
    On production servers, module loading should not be necessary, because the hardware should not change frequently. Also, with today's computers, kernel recompilation times are normally low[6], and many options can be directly compiled, as kernel size should not be a problem anymore. Thus, it

[5] an example of this kind of rootkit, completely available with full source code, is the *Adore* module at team TESO web page `http://www.team-teso.net`.

[6] about 20 minutes on a Pentium 4 @ 2.5 GHz.

is recommended to disable this feature whenever possible to strengthen a working system.

- Disable writing to /dev/*mem, or other raw memory access calls.
  - The ability to read or write directly to any memory location can possibly modify anything in the system, so unless absolutely necessary this should not be possible, even for the root user. Under Linux, the */dev/mem, /dev/kmem* devices allows such a modification and while it is technically difficult to exploit, it is nevertheless a weak link. Other ways to achieve the same goal exist, such as */dev/microcode* and the *iopl()* call. We completely disable the existence of these modules with some patches for the 2.4 Linux kernel series, or use the newer security modules under the 2.6 Linux kernel.
    The main problem here is that there are some programs which need those devices to work correctly, the most famous of them being the *X* window system, and it can be difficult to make them work without those devices. Under production servers, where *X* window usage should be rarely needed, a workaround is to use remote *X* servers whenever needed.
- Make raw hard drive devices read-only.
  - This is necessary to disallow direct circumvention of software file protection. Else, an attacker could modify the files by issuing commands directly to the hard drive, and then reboot the server in a less secure configuration.
- Use special file-system capabilities.
  - Modern file-systems (coupled with an appropriate operating system) are able to force severe restrictions for all the users, and also potential intruders. This is the case of OpenBSD and Linux append-only file flags, which cannot be turned off under normal circumstances, sometimes not even by root. This will make hard for an intruder to modify the configuration to get

rid of some security features, and reboot the the system in such an insecure state.

- Disallow execution of untrusted files via trusted path execution.
  - To make it more difficult for an attacker to compile and execute his local exploits, it is possible to force the system only to run programs which are inside root-owned, non-writable directories. This makes privilege escalation within a machine very difficult, as it does not allow newly generated code to be executed. This kind of protection is called Trusted Path Execution (TPE), and is currently being developed as a Linux Security Module; it is also available as a patch[7] for OpenBSD.
- Make some parts or binaries of the operating system read-only.
  - This makes permanent system and configuration files modifications impossible. It can be achieved by booting from a CD-ROM or some other kind of device which does not allow writing, like a solid state disk with write protection.
- Patch the kernel and programs to implement various buffer overflow, stack protection mechanisms, and other security features.
  - This strengthens the system against unknown attacks. These kind of tools are not yet mainstream, but they are increasingly gaining popularity. For compiling safer programs with GCC, we have *propolice*, [8] and *stackguard*[9]. Both of them are GCC stack smashing protection programs. Programs compiled while using this GCC extension are much harder or impossible to exploit via remote stack smashing bugs. More options include using *PaX*[10], which addresses code injection within a working program, and adds address space layout

[7] called Stephanie, although it is a bit outdated.
[8] Propolice web site (IBM): `http://www.trl.ibm.com/projects/security/ssp/`
[9] Immunix web site: `http://www.immunix.org/`
[10] Page Exec: `http://pageexec.virtualave.net/`

randomizations using ELF shared objects as executables, effectively making automated attacks very difficult and much more likely to be discovered, as exploitable bugs will have a high chance of crashing the program instead of making it run the external unwanted code.
Some of the biggest and more developed system modifications in this category are LIDS (Linux Intrusion Detection System), GrSecurity, and SELinux (Security Enhanced Linux). We will review them in the following sections, and actually implement and configure a SELinux-based test machine for this thesis.

There also already exist a good number of system utilities whose purpose is to analyse a PC searching traces of possible successful intrusions, by means of monitoring accesses and changes to critical files or processes. A not exhaustive list of these utilities, known as host-based agents, include Tripwire, Axent, CyberSafe and ISS.[11]

Other utilities and possible important procedures that we need to keep in mind if we want to set up a secure server with Linux, and are explained in detail in other papers and books, like [Bau02].

We will now analyse a well deployed and production ready security architecture example, present in the OpenBSD operating system. After that, we will analyse the three mentioned big software solutions which are actually being developed for Linux.

2.3 An example implementation: OpenBSD securelevels

One of the best possible widely deployed examples of system lockdown is present in OpenBSD. OpenBSD[12] has implemented a number of security measures directly within its operating system, effectively disallowing certain operations to be executed even by root, in what they call "*securelevels*". A securelevel is a special kernel flag, ranging from -1 to

[11] Tripwire is the only of these programs which has an academic license for universities, all the other are commercial programs. They can be found at www.tripwiresecurity.com, www.axent.com, www.cybersafe.com and www.iss.net

[12] `www.openbsd.org`

2, and which can only be *raised*. Higher securelevels imply harder restrictions over the system, and as securelevels cannot be lowered, once a restriction is in place it cannot be revoked, not even by the superuser. If a legitimate root wants to perform tasks which need a lower securelevel than the one being used, he has to reboot or put the machine in single-user state[13] (which, among others, disables networking) and be physically in front of the machine. Thus, an intruder cannot remotely modify the protected machine settings.

The text below shows the securelevels' man page from an OpenBSD system summarizes all the restrictions:

```
SECURELEVEL(7) OpenBSD Reference Manual SECURELEVEL(7)
NAME securelevel - securelevel and its effects
SYNOPSIS The OpenBSD kernel provides four levels of system security:
-1 Permanently insecure mode
   - init(8) will not attempt to raise the securelevel
   - may only be set with sysctl(8) while the system is insecure
   - otherwise identical to securelevel 0
0 Insecure mode
   - used during bootstrapping and while the system is single-user
   - all devices may be read or written subject to their permissions
   - system file flags may be cleared
1 Secure mode
   - default mode when system is multi-user
   - securelevel may no longer be lowered except by init
   - /dev/mem and /dev/kmem may not be written to
   - raw disk devices of mounted file systems are read-only
   - system immutable and append-only file flags may not be removed
   - kernel modules may not be loaded or unloaded
2 Highly secure mode
   - all effects of securelevel 1
   - raw disk devices are always read-only whether mounted or not
   - settimeofday(2) may not set the time backwards
   - pfctl(8) may no longer alter filter or nat rules
   - the ddb.console and ddb.panic sysctl(8) variables may not be raised
DESCRIPTION
Securelevel provides convenient means of "locking down" a system to a degree
suited to its environment. It is normally set at boot via the rc.securelevel(8)
```

[13] *init* is the only process able to lower the securelevel, and does so when entering single user level.

```
script, or the superuser may raise securelevel at any time by modifying the
kern.securelevel sysctl(8) variable. However, only init(8) may lower it once
the system has entered secure mode. A kernel built with option INSECURE in the
config file will default to permanently insecure mode.
Highly secure mode may seem Draconian, but is intended as a last line of defence
should the superuser account be compromised. Its effects preclude circumvention
of file flags by direct modification of a raw disk device, or erasure of a file
system by means of newfs(8). Further, it can limit the potential damage of a
compromised “firewall” by prohibiting the modification of packet filter rules.
Preventing the system clock from being set backwards aids in post-mortem analysis
and helps ensure the integrity of logs. Precision timekeeping is not affected
because the clock may still be slowed.
Because securelevel can be modified with the in-kernel debugger ddb(4), a
convenient means of locking it off (if present) is provided on highly se-
cure systems. This is accomplished by setting ddb.console and ddb.panic to 0
with the sysctl(8) utility.
FILES
/etc/rc.securelevel commands that run before the security level changes
```

To secure the system logs as strongly as possible, a sysadmin using OpenBSD should do the following:

- Put immutable flags on important */etc* files, and append-only flags on log files, typically located in */var/log.*
- Set *ddb.panic* and *ddb.console* to 0.
- Raise the securelevel to 2

Having a high securelevel of course also makes the administrator life harder, as he cannot load kernel modules, change some system files, nor touch hard disk partitions. On the other hand, on a stable production system, the actions below should be rarely performed by a legitimate admin, while they are crucial for attackers:

- delete or modify system logs and configuration files through the operating system or directly through raw device access.
- change the system clock.
- load kernel modules to change the behaviour of the system.

Of course, this is a software only solution. As most software only solutions, if an attacker finds a hole in the implementation of these measures, he will be able to circumvent them. But the OpenBSD team is very devoted to computer security and performs continuous code audits on their code, resulting in very few security advisories compared to other operating systems. In fact, while working on the present thesis, the latest vulnerability I was been able to found affecting securelevels was back from 1998[14]. Just recently a new weakness just appeared[15], dated September the 14th, 2003, which allows semi-arbitrary writing of the kernel memory within a high securelevel state.

2.4 SELinux

Secure Enhanced Linux is a complete set of modifications to the Linux kernel, developed by the American National Security Agency, which allows strong security measures in the systems which implement it. The security mechanisms are implemented at kernel level. Along with the patch, there are several userland tools to interact with the implementation.

Security Enhanced Linux (herein SELinux) kernel code is currently implemented as a set of patches in the 2.4 and 2.5 Linux kernel series. For the 2.6 series, it has been heavily modified to be able to fit within the Linux Security Modules architecture, and will be incorporated into the base kernel. This will make SELinux much better known by the average Linux user and system administrator. We hope such an increase in public awareness will help both the code development and the number of production machines using it, and thus make a hopefully safer Internet.

2.4.1 SELinux security model

SELinux implements two security approaches at the same time, which interact with each other. On a lower level there is TE (type enforcement),

[14] 4.4 BSD mmap vulnerability `http://www.openbsd.org/advisories/mmap.txt`

[15] Securiteam.com advisory: `http://www.securiteam.com/unixfocus/5OP0E0AB5E.html`

and using TE is a RBAC (role-based access control) architecture. Let's see what are both of these solutions, and how they interact.

Type Enforcement with Role-Based Access Control. We can abstract a computer operating system like a collection of objects and processes, interacting with each other. Objects are files, directories, sockets, etc., which would be used by processes. For every object and process, we will assign a certain number of security attributes, and form a security context. The security context itself does not enforce anything, but will be interpreted by the security server. Whenever a security decision is needed, the security server will take the security context of both objects or processes interacting, along with the exact class of object (file, directory, process) which has triggered the decision code. It will then use all this information to deny or allow the interaction taking place.

For dynamic structures and processes, the policy enforcement will assure that security contexts are generated on the fly according to the system configuration. On the other hand, persistent objects will need to have their security contexts stored to disk, after being initialized with the *setfiles* utility. Note that file context configuration is separated from the policy configuration. The former is only used by the policy enforcing code, while the later is used by the the security server.

The two security contexts, the object type and the policy configuration are merged by the by the security server to generate an access vector. This access vector states all the allowed and disallowed interactions between objects, and will be passed to the policy enforcement module, which will cache the information for better performance. The security server will also tell if we want the event to be logged, according to our policy.

This seemingly complex architecture allows a very fine grained modelling of how we expect our system to behave. As all the accesses to objects are checked with the policies, we can assure work-flow pipelines, and full separation of applications. We can limit the damage a malfunctioning or hacked application can impose to the system, thus protecting the information.

This goes well beyond the use of the standard Linux Discretionary

Access Control (DAC), as every piece of the operating system can be finely described. As an example, the files owned by the web server user do not all serve the same purpose. They can be web pages, directories, programs, scripts, templates, links, logs. On a standard Linux security model, all these files are owned by the same web user, and could potentially be modified by it. With SELinux we can differentiate all these files, and create different rules for them, while still being "owned" by the web server. This works transparently, and the server will not be disturbed by the SELinux system unless he tries to perform some unexpected action. Furthermore, not only standard Linux files can be described this way: network sockets can also be labeled and controlled within SELinux.

Access on files and different devices also go beyond the normal Linux DAC. On files, it is possible to make rules controlling the creation, reading, deletion, append, seeking, and more. Processes can be disallowed the use of shares libraries, locking, IPC, etc. Network can be controlled independently of the system firewall. These and many other interception points placed within the kernel help confine any program in a place where it can only perform the task it was designed for.

2.5 GrSecurity

GrSecurity is a suite of patches to the Linux kernel implementing a big number of security measures. Its main feature is providing a system-wide Access Control List (ACL), able to control very precisely what every system process is permitted to do. Using it, access to files, capabilities, resources and sockets can be granted or denied to all users, including root. GrSecurity also includes PaX protections, which are designed to protect the system from buffer overflow and stack smashing bugs, the two most commonly used techniques to infiltrate a system, and also has measures to prevent processes from consuming all the system resources.

Just like SELinux, the protecting code runs directly within the kernel, and is controlled by a number of dedicated userland utilities. The ACL it implements is not coupled with a role-based model, which make it less flexible than SELinux's one. But on the other hand, implementing PaX protections makes it a good alternative. Summing up, both SELinux an GrSecurity offer a big increase in the system security by allowing a very detailed description and containment of program capabilities.

2.6 Other kernel patches

GrSecurity and SELinux are not the only available tools to implement a safer Linux. Similar to GrSecurity, the Linux Intrusion Detection System also implements ACLs, along with special features like a port scanning detector within the kernel. All these three systems are actively being developed for the soon-to-come 2.6 Linux kernel, whose security module architecture helps the integration of external security frameworks. The only drawback is the increased configuration time needed to set up a system correctly, which will make these features unattractive to the average computer user. But serious system administrators will find soon find themselves with new, powerful tools to set-up their system in a more secure way.

Chapter 3

System logging

3.1 Introduction

Properly storing the information originating from a whole system in an organized way is automated via *logger* programs. Normally only one of those is running at a single time, and acts as a middleman between the data it receives and whatever use we want to give to that information. Via system calls, locally running programs send the data they want to log to the logger. Then the logger uses its knowledge about the program originating the information and the type or importance of that information to process the message in whatever way we tell it to. In the simplest configuration, we will simply store the messages in the hard drive for a determined period of time. But any kind of processing is possible: we can e-mail specific informations, create reports, encrypt the data, send it to a remote host, or many other possibilities. The flexibility of modern system loggers make them a very important piece in the set-up and configuration of computer servers.

3.2 Logging under Linux

The traditional linux system logger is called *syslog.* This program, originating from the old Unix servers, is old and has been improved. More modern replacements are *syslog-ng, metalog* and *msyslog.* All of them are faster and offer the same or more functionality than his precursor.

The set-up of these daemons is generally simple. Albeit using different semantics, they all perform similar tasks, filtering the input to the output as determined by their configuration file. We do not pretend to explain their configuration in detail, as comprehensive documents about Linux loggers are freely available on the Internet. If specific details are needed, the reader is recommended to check *Building secure servers with Linux* [Bau02] chapter about system log management[1], where operation of *syslog-ng* is explained.

Normally the system loggers only perform a limited amount of processing on the data, leaving complex tasks to dedicated programs. The most usual program is the *logrotate* script, which periodically renames and compresses the log files, to save space and limit the amount of logs we store. Many other exist, specialized in the information coming from specific sources. These log analyzers are responsible of generating alarms any time a suspicious activity is found. Two well known are *swatch*, and *logsentry,* which can scan and perform various actions depending on the output log data. These actions include sending alarms to the system administrators in case a security breach or any other anomaly is detected.

3.3 Remote logging

Probably the safest mode of operation for a server is not to store its own logs locally, but on another host. This is achieved by sending the log data over the network to a centralized log server, which will store the data in an environment specifically designed for security. All the *syslog*-like programs are capable of this, and this set-up if often used in large environments where logs are important, with money to invest in a machine specially dedicated for log storing.

[1] which is freely available on-line at Oreilly: `http://www.oreilly.com/catalog/bssrvrlnx/chapter/ch10.pdf`

This schema works perfectly in a controlled environment where the network is not saturated and packets are not lost. Traditionally, UDP packets have been used for log transmission. TCP has become an option with the newer loggers, and is safer against lost packets. On the other hand, UDP has a lower overhead, and does not block its operation when the communication links are saturated.

As the data travels over the net, it is important it is not captured by other hosts. To assure log privacy, the companies can use private or dedicated networks if available, encrypt the data being sent, or both. It is also possible to keep the listening log server address unknown, by broadcasting the encrypted logs over the net the server is listening to. Anyway, this kind of infrastructure may not be always available, and budget will ultimately decide if we use this configuration.

3.4 Base logging speed

Before trying advanced features which will make use of cryptography to secure the system logs, speed measurements of the different modern *syslog* replacements has been performed. These are the average results obtained.

Machine: P3-866, 256 MB of RAM.

Input used: 1 million different 55 bytes-long average log entries (54 MB)

	syslog-ng	metalog	msyslog
Time	39.8	41.5	111.8
KB/second	1371.3	1315.2	488.2
Log entries/second	25125	24096	8944.5

As we see, *syslog-ng* and *metalog* are very fast, while *msyslog* is the slowest of the three. But *msyslog* has a feature which is very interesting for secure log storage: it can authenticate and encrypt the logs it generates.

3.5 PEO and L-PEO

Msyslog can add authentication to the log it generates. It does it by safely storing an initial value, K_0, and creating a hash chain based on both the log data it receives and this key. This is the PEO algorithm:

- Generate a random K_0. Store it in a safe place (method not implemented by msyslog)
- Repeat : when new data $D_i$arrives, generate and store K_i =hash (K_{i-1},D_i)
- Delete the previous K_{i-1}.

This allows the person in possession of $K_0$to recreate the the hash chain, and verify if the K he computes from the logs is the same which is stored on the machine. If it is, then the logs have not been tampered. L-PEO is identical to PEO, except that it adds a Message Authentication Code to every stored line, using the previous K as a key. This allows the system administrator to know which line was modified in a corrupted log.

We will examine in detail the hash chains and their benefits in the CryptoHasher chapter. Meanwhile, we can see the impact enabling this features cause on the msyslog throughput, using the same test bed as the previous benchmark:

	msyslog	msyslog+PEO	msyslog+L-PEO
Time	111.8	299.2	359.4
KB/second	488.2	182.5	151.86
Log entries/second	8944.5	3342	2728.4
Relative difference	100%	267%	321%

As we can see, the performance penalty under the current implementation of PEO and L-PEO is quite important, specially if we note that the hash used is the fast MD5 algorithm. Also, these protocols do not specify how to safely store the initial K_0. We will address this, and further explore the possibilities of applying cryptography to the system logs in the next chapter.

Chapter 4

CryptoHasher

An implementation of Bruce Schneier's algorithm.

4.1 Introduction

In a paper dating 1997 [SCH97], Bruce Schneier defines an approach to add authentication and encryption to system logs. The main benefit is impeaching a successful attacker to read or modify previous log entries undetected. This setup only needs an initial interaction between our attacked machine ("U", for untrusted) and a trusted ("T") machine. Then, periodically, T, or, with T's assistance, another server, could check the integrity of the logs and decrypt them as needed. Also, thanks to the key creation scheme, T could send copies of those logs to other moderately trusted servers, and allow selective decryption of them.

Following are the details about the algorithm and discussion of its benefits. After that we will explain and discuss the actual implementation we did under Linux. Finally we will see how we can integrate this program with a SELinux secured machine.

4.2 Hash chain as means of key generation.

The algorithm builds on top of the one-way feature of hash algorithms. Given a hash value, a secure, irreversible hashing algorithm makes it very difficult to find original data which computes to the same hash. We are going to use this property to create a hash chain: a succession of values in which every value is the result of hashing the previous one. The original, first value, will be called A_0, successive values will be called A_1, A_2, ..., A_n, any value of A_n may be used to calculate values A_{n+m} ,but not previous values. As we store the original A_0in some safe place, we will locally be using one after another the elements of the A hash chain, but make sure that we only keep in memory the last $A_n$value. If we only keep this A_n key, we will be sure that previously encrypted data, D_m, will only be accessible either by:

1. Having the used A_m or any previous A_l, with $l<m$
2. Computing that A_m, $m<n$ from A_n.

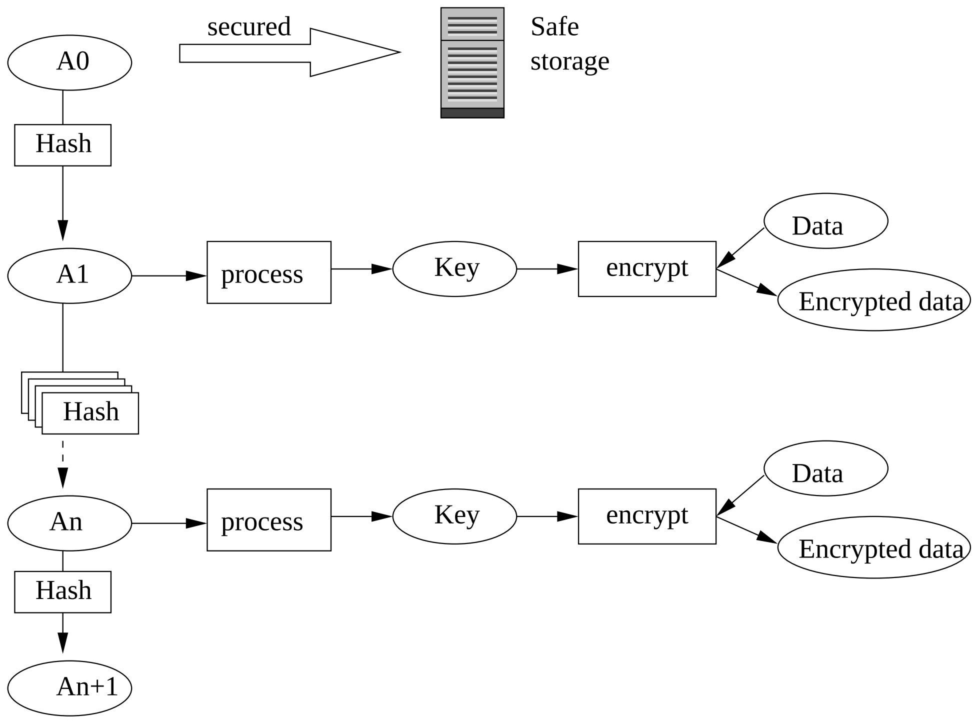
But by choosing secure hashing algorithms, we ensure that (2) is not possible, so the only way do decrypt previous data is by having access to the only A stored: A_0, which is not stored in the local machine. This is true for both a legitimate system administrator and a successful intruder, no matter what control they may have over our machine. Thus, after storing the original A_0in a remote computer, each data (in our case, a log entry) encrypted based on A_n is locked as soon as we overwrite it with the next hash iteration, A_{n+1}.

This method keeps the $E_n$encrypted data unreadable if we do not know the current or previous values of A_n, but does not offer authentication of the data. Thus, some extra work is needed if we want to be sure there has been no modification of the data. Also note the box called *process* on the figure 4.1, we will discuss about it on section 4.4.

4.3 Hash chain used for authentication

To be sure of the integrity of the previous data, we add another hash chain: Y. This chain is calculated in a different way than A's chain:

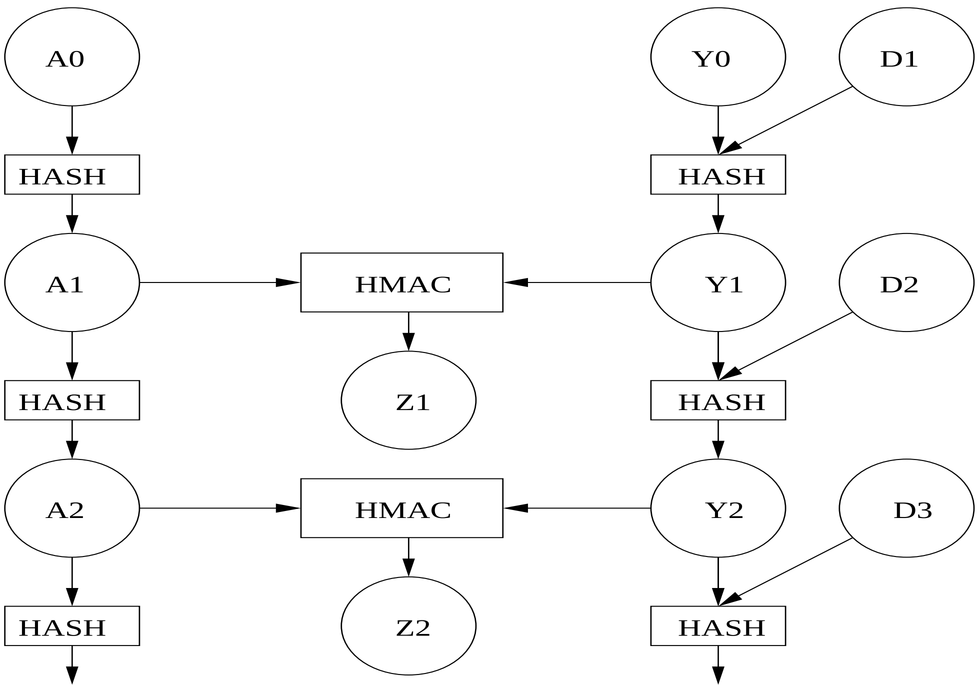
Figure 4.1: General approach to using a hash chain for encryption



before hashing the current $\mathrm{Y_n}$value, we append to it the data we want to add integrity checking to, making $\mathrm{Y_{n+1}}$=hash $(Y_\mathrm{n}, Data)$. What we achieve with this is a chain that depends of all the previous data. This can detect single deletion of log entries, but is not enough for authentication, as an attacker could delete some entries and recalculate a valid Y chain.

To solve this limitation, we will sign the hash using a symmetric message authentication protocol, like HMAC. For this purpose, we will use the previously defined A hash chain, signing every $\mathrm{Y_n}$with the corresponding $\mathrm{A_n}$. The result is a pair of $\mathrm{Y_n}$,$\mathrm{Z_n}$along with every $\mathrm{D_n}$log entry (see figure 4.2). To authenticate the full log, all we need is to recreate the D-Y hash chain for the stored data, and verify that the last signature is correct. We could verify each entry, but it is not necessary, as verifying the last signature is essentially verifying each previous log entry.

Figure 4.2: Double hash chain used for authentication



4.4 Adding log type information

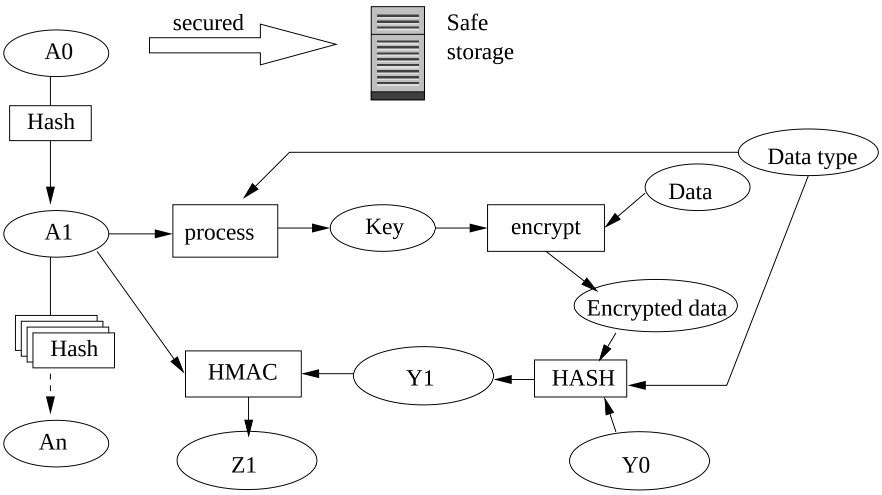
In sections 4.2 and 4.3 we use separated solutions for adding encryption and authentication. Our purpose is to integrate both of them, but before that we can add another feature to the algorithm: log type differentiation. As each log entry may contain data of different levels of confidentiality, it would be interesting if we could separate them before encryption. This would allow us to give out the full encrypted log to a third party, and allow the third party to decrypt individual log entries, but not the full log.

To perform this, we add log type information to every log entry. We will call this information W. We can use W to influence the box called *process* on figure 4.1. If this *process* is $K=hash(A,W)$, then we are effectively generating a different key for every log entry, and that key is different from A_{n}. Note that due to the one-way nature of secure hashing algorithms, it is impossible to retrieve $A_n$from a given K. Also, the hash chain has to be modified to include the log type information, to make a new hash we use Y_{-1}, W and the encrypted data E.

What this approach allows is to give the full logs to a third party and allow that third party to read only limited parts of the log. This third party, which we will call V, would interact with the trusted server (herein, T) keeping $\mathrm{A}_0$and ask him the $\mathrm{K}_{\mathrm{a,b,c}}$keys corresponding to several pairs of $\mathrm{D}_{a,b,c}$,$\mathrm{W}_{\mathrm{a,b,c}}$. With this information V can read, but not modify, individual log entries. Authentication is also possible: the third party would send his last pair of Y_{n},$\mathrm{Z}_{\mathrm{n}}$to the server. If the server acknowledges, by using A, that the signature is correct, V is assured of the integrity of the logs. Note that in this interaction the trusted server does not need to receive the full log archive, nor know the data D. The third party will send him the indexes of the entries he wants to read, along with the type of data represented. Then the trusted server would reply the K corresponding to those indexes, using his knowledge of A and the received data type.

Thus, if we have two kinds of entries, *normal* and *confidential*, and V is only allowed to read *normal* entries, he cannot lie to T and ask him for some log entry indexes of *confidential* type. He would get some keys back, but those keys would be $K_n=hash(A_n,normal)$, instead of $hash(A_n,confidential)$. The key received would be wrong, and V will not

Figure 4.3: Algorithm including log type data.



be able to read the confidential log entry, nor he will be able to guess the correct key from the received one.

4.5 Implementation of the algorithm.

For this project, we have implemented the server and client part of the CryptoHasher under Linux. We wanted the system to be easy to integrate under existing Linux environments, and tried to limit the amount of computing resources used. We have thus chosen C as the language due to its speed. For low level encryption, we are using the *Libtomcrypt* library, as is it quick and supports the latest algorithms. For the high-level certificate management and SSL connection, we have used the *cryptlib* library, which offers powerful high-level operations. Thus, we expect to have achieved and implemented the following requirements:

- Defaults to transparent input to output (*stdin* to *stdout*) processing of data. This allows the program to be easily used as an intermediate pipe at the output of any program.

- Support for raw binary input and output of data. Testing includes several hundred megabytes of random data proceeding from

/dev/urandom being encrypted, authenticated and stored, decrypted and verified[1], and text lines from 1 to 100.000 characters.

- Encrypting and hashing algorithms are choose-able, so users may select the best performance-security trade-off for their application.

- Encryption and authentication added to the data makes it useful as an intermediate step for storing slowly growing sensitive information in hostile environments.

- After the initial setup involving the master server, the system may be disconnected from the network. As long as new logs are not opened, network presence is not necessary.

- Limited network bandwidth requirements make it useful in bandwidth-limited situations.

- Authenticated and encrypted connection with the trusted server, using industry standard X.509 certificates.

4.6 Technical information

4.6.1 Detailed encryption and hashing method

The algorithm used for encryption works as follows:

1. Upon opening of the log file, a master key (A_0) is randomly generated and committed securely to the trusted server. After the trusted server acknowledges the transaction, an initial entry is added to the log, and logging can begin. If the trusted server does not answer normally, we write an "abnormal close" message, and the log file is closed.
   This step is important, as we want to be sure an attacker cannot

[1] Note that /dev/urandom does not provide a perfect secure random input as /dev/random does. But using /dev/random is very slow, as it gathers entropy from different parts of the linux system which is a slow process. As our objective here was to test the CryptoHasher suite under any variety of random binary input, /dev/urandom suits our purpuse.

delete files and then claim they never existed. If the full transaction with the trusted server is stored not only on server side, but also on the client side, it is impossible to delete a log file and pretend that the client never received the answer from the server. Every connection received by the server, successful or not, must thus have a full record on the client side.

2. New logging data (D) is read from the system logger, along with it's type (W).[2]

3. A new key (K) is derived from hashing the concatenation of A and W, overwriting the previous K (if it exists).
   This key is not susceptible to dictionnary attacks, as the result of secure hashes are elements within a n-bit[3] space address, pseudo-randomly located. An attacker would need to explore (half) the full hash space address to find the key[4], which would be absolutely unpractical with todays computers, as long as we use a hash with a long enough output. To cite only some of them, the relatively weak md5 algorithm output is 128 bits, the widely deployed SHA1 algorithm output is 160 bits, and the newer and more secure SHA-256 output is 256 bits.[5] Thus, the key part of the encryption is safe: given the encrypted data, the attacker would have to concentrate attacking the encryption algorithm, trying to guess the key would not be feasible.
   It is important to note that because of W being stored in cleartext, the attacker also knows part of the string (A,W) used to create K. So instead of trying to guess K, a better attack would be to try to guess A, and use it along with the W value to create series of Ks and see if the key works. Both A and K have the same number

[2] Note that in the present implementation, the log type is fixed via a parameter when CryptoHasher starts up. In future versions we expect to parse the first characters of the input searching for some token indicating the log attribute type.

[3] where "n" is the number of bits at the output of the hash

[4] A normal measurement of an algorithm resistance is how many addresses we would need to explore to have a 50% probability to find the key. The worse-case scenario for an attacker would be a brute-force attack, trying every possible key. In this case, half of the total possible keys would have to be explored.

[5] There also are versions of higher length: SHA-384 and SHA-512.

of bits, so the attacker would probably better try to guess A. It is advantageous to find A, as it could be used to decrypt (and modify!) all the following records. On the other hand, we would need to perfom an additional hash for every key tried, slowing the speed of the attack. As normally hashing speeds are faster than encryption speeds, this would not be a big problem, adding a linear % extra time to the processing, but slowing it nonetheless.

4. This new key is used to encrypt the data (D) using a symmetric cypher. We will call the encrypted data "E".
   Here the length of the key used is the same as the output of the hash function, so unless extra processing is performed, the cipher has to accept key lengths of the same size as a hash value. Using the output of the hash directly as the key can thus be performed on certain combinations of hash and cipher algorithms only. SHA1 is specially problematic, as its 20 byte output can only be used on the Blowfish and RC2, RC5 or RC6 ciphers, as they accept a variable key size. Other working combinations include, but are not restricted to, MD5, SHA256 with AES, Twofish or Blowfish.Anyway, as the used key is random and at very least 16 bytes (MD5), or easily 32 bytes (SHA-256), only flaws in the encryption algorithm would be able to threaten this step. All of the proposed algorithms in this thesis are believed to be secure by the international community. Also, the user has the freedom to choose any pair of algorithms in case of any flaw being discovered in the current ones. The user should also be aware that as security and number of bytes increase, the processing time will too, so he has to make a choice regarding increased security versus performance tradeoff.

5. A new hash is made (Y) out of the concatenation of the previous hash (Y_{-1}), E and W.
   This step is the base for the message hash chain. The only possible attack here would be finding a different message, (Y_{-1},X,W) which would generate the same output. As $Y_{-1}$and W are known by the attacker, he would only be able to play with a modified E. But it is very unlikely that an attacker would be able to find a collision; it may be possible with a huge computation power for the less secure

hash algorithms, as MD5, but certainly not for SHA256. And in the case he succeeds, he would only replace the original E with a bogus X value in the log, but still would not be able to read the original E.

6. This new hash is authenticated using a symmetric message authentication code, HMAC, which uses A as the authentication key. This authenticated hash will be called Z.
   This step protects the hash chain integrity; without it, it would be possible to modify any Y entry an recreate all its values. In our implementation, the HMAC algorithm internally uses the same hash algorithm as the other parts of this program, but it could be modified to be used with another hash otherwise.
   Using the same master key for authentication and encryption would completely break our scheme if there was some kind of attack successful at finding the key A from the pair of available Y and Z values. Fortunately, such an attack is currently unknown. But the HMAC algorithm[6] is only as strong as the hash used by it, so for the security of the log it would be preferable to use SHA1 or SHA256 keys, as MD5 has some weaknesses and is assumed as a relatively weak secure algorithm nowadays.
   The only drawback of using this approach is that the authentication can only be verified by re-creating the A value that was used to sign the hash. This means that only verifying the hash chain is not enough to ensure the log's integrity, as the attacker could have modified several records of it and re-created a new, different, valid Y chain. Without the A value to verify it, the client or a third party cannot authenticate the log. Only by verifying the signature we can be sure of the hash chain's integrity. That's why if a third party wants to verify a log's integrity, he will have to send the Y_n,$Z_n$pair to the trusted server.

7. A is incrementally hashed, its result overwriting the old A. Once this step is finalized, we cannot read nor alter the data anymore.

8. We store W,E,Y and Z in the log file.

[6]HMAC=hash(key,hash(key,data))

9. We are ready for new data, back to step 2.

10. Closing the log file implies writing a log entry with type (W) = NormalCloseMessage, and a time stamp as data, (D) which is normally processed. A and K are also wiped so they cannot be recovered.

4.6.2 Internal data structure

The first message in a log is special, as it records the transaction done with the trusted server. Encrypted inside it is the value of the initial A_0, along with the date of the log's creation, and an unique ID for this log. Its uses the following format:

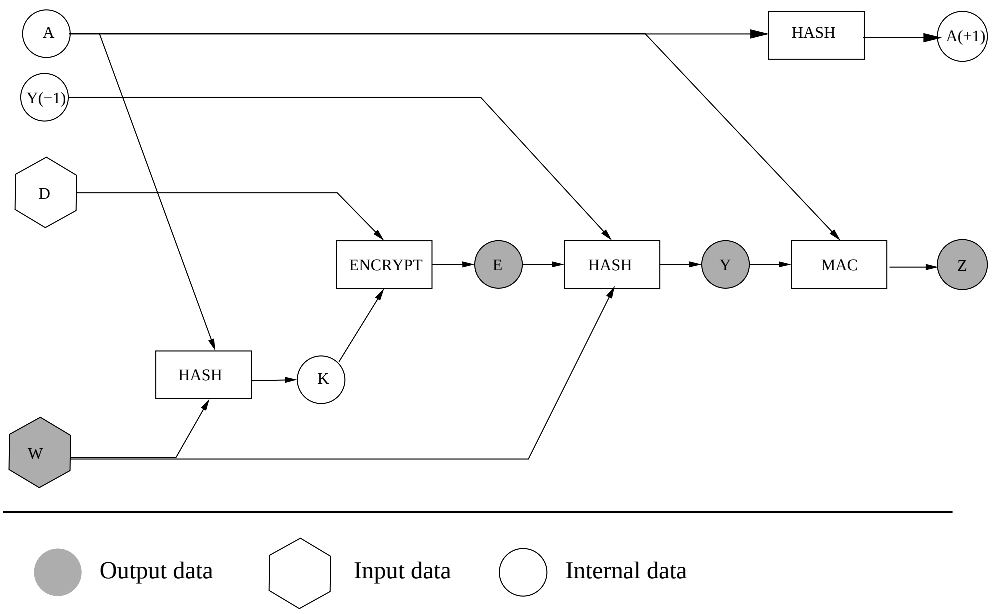
- W: "LogFileInitializationType", padded with 0's until size SIZEOFFACILITY.[7]
- unsigned long timestamp (4 bytes): timestamp of log file opening.
- unsigned long timestamp (4 bytes): timeout for waiting the trusted server answer.
- 64 bytes: unique ID for this log, generated by the machine, of size equal or less than IDLOGSIZE.
- unsigned long: size of M0, the log opening message sent to the trusted server, which we will discuss in the next section.
- binary data: M0, message sent to the server upon opening the log file.

After that, the CryptoHasher protocol version 1 begins its main data processing rutine, and stores all the subsequent data in the log file using the following format:

- unsigned long (4 bytes): length of W.

[7] global variables, which are indicated by all-capital leters, can be modified from the *hasher_common.h* source file, and will affect all the programs in the CryptoHasher suite.

Figure 4.4: Data path followed by the different Cryptohasher variables to write a log entry



Variable	Use
A	Master key (secret), used to calculate the current password and authenticate the hash chain
Y(-1)	Previous value of the hash chain
D	Unencrypted data we want to store on the log
W	Type of log we are storing
K	Key used to encrypt the current data
E	Encrypted log data
Y	Hash chain of the current data
Z	Signature under A key of the current hash chain Y.
A(+1)	Hash of A, must overwrite its previous value

- string: W. Max size of W determined by global variable SIZEOFFACILITY.
- unsigned long (4 bytes): length of E.
- binary data: E. Max size of E determined by the global variable SIZEOFBUFFER.
- binary data: Y. Hash chain; with the default hashing algorithm (SHA256) it's size is 32 bytes.
- binary data: Z. Hash chain MAC, of the same size of Y.

The first entry with this format in the log file should be a ResponseMessageType one, with the answering message (M1) from the trusted server as encrypted data. This ensures that the initial interaction with the trusted server was correct, and thus logging may follow.

The last entry in the log should be of type NormalCloseMessage, with a timestamp as data, formally indicating that the log is closed.

In case there is some problem interacting with the trusted server, there should be an AbnormalCloseType message, with a timestamp and the reason for closing the log file. This assures non-repudiation of a communication, so that some intruder controlling the untrusted machine cannot delete a logfile and pretend the machine did not receive confirmation from the server.

4.6.3 Interaction with the master server: messages sent

In the interaction with the server, the client sends the following message (M0) to the server, wrapped under a SSL session. The SSL session is a simple username / password session, which adds an outer envelope to keep transmitted data out of possible eavesdroppers, and increases the challenge of starting any kind of interaction with the trusted server. Under the SSL layer, we will use X.509 certificates to authenticate both the client and the server. The machine's authentication is based a trusted Certificate Authority (CA) signing both the client and the server's certificates. The client will send it's certificate to the server, and if the server acknowledges the certificate as having a valid signature from the trusted CA, the interaction will continue.

These are the messages exchanged between the client and the server. First the client sends this message, which we will call M0

- 32 bytes: Protocol: "hasher_protocol_v1"
- 16 bytes: encryptor: name of the encryption algorithm used for the rest of the log file.
- 16 bytes: hash: name of the hash algorithm used for the rest of the log file.
- 64 bytes: unique identifier of the client machine, as it appears on its certificate.
- unsigned long (4 bytes): length of the initial data message, X0.
- binary: X0 data message encrypted with the server's public certificate, and signed by the client's private key.

where X0 is composed of:

- 32 bytes: Protocol: "hasher_protocol_v1" is the name and version of the protocol used in the current configuration.
- 4 bytes: timestamp of sending time
- binary data: random initial A0, size according to hash used, which is 32 bytes in default (SHA256) configuration
- binary data: random initial initialization vector (IV) for the Counter Mode (CTR) chaining used for encryption.
- 2048 bytes: space reserved to put the client's machine certificate. It must be signed by a root CA authority acknowledged by the trusted server.
- 512 bytes: space reserved for the signature, under the client's private key, of all the previous data in X0.

The client will keep a local copy of X0's hash, and record the first entry in the log message. This entry will be of a *LogfileInitializationType* log type, and store the current timestamp, timestamp timeout, client-side generated ID log, and the full M0 message sent to the server. As the M0 message itself is stored encrypted, it can be used to decode the log file by using the server's private key. But this key is of course unknown for the client machine or for external intruders; keeping that first message is not a security threat as long as the X0 message it contains is properly encrypted with the server's certificate.

Storing the first initial message in the log file can also serve another purpose: creating new log files while the client cannot connect to the server. In this case, the client would not wait for the server's answer, and begin logging messages while unconnected. After getting access to the server, he would send him all the unsent log opening messages. There is a drawback to this situation, as we are not protected from undetected full log file deletion in the case the client was exploited before he connects to the server. But if the client has some write-once media available, big enough to store those initial messages while unconnected, then this is not an issue anymore.

The IV sent is kept the same during the whole log duration. This is not really important, as an attacker could only take advantage of this knowledge if analyzing several CTR ciphered messages with the same IV and the same key. But this is not the case, as our key is changing with every log entry. Thus, mantaining another hash chain for the IV would be an unnecessary waste of resources.

Once the initial message has been sent, and the temprary data deleted from the client, only the server can decrypt the X0 message and verify it was correctly signed by the client. If the certificate and signatures are valid, he sends this reply (M1) back:

- 32 bytes: protocol.
- 64 bytes: unique identifier of the trusted machine, as it appears on his certificate.
- unsigned long (4 bytes): length of the reply data message, X1, which is going to be sent to the client.

- binary: X1 data message , signed by the servers private key and encrypted with the client's certificate key.

X1 is composed of:

- 32 bytes: protocol.
- 64 bytes: unique ID for this log, sent by the server.
- binary: hash of the original message data, X0.

Thus, the client can check the server has received its message and decoded it correctly (the client keeps a copy of X0's hash for this purpose). If everything is correct, he will write the second entry in the log, with W=ResponseMessageType and M1 as data; once this is done, the client will start processing log entries submitted to it. All the interaction takes about two seconds on two computers on the same LAN.

We have to note that both the client and the server create a unique log ID, but the client machine does not send its local log ID to the server. Then, the server sends a new id back to the client, which stores it, but in encrypted form. This allows the server to choose and keep secret his own identification scheme for the files, independently of the client. The uniqueness of the log files identification is still assured, the server only needs to record the pairs of X0(A0) and his log ID answer to keep a record of all the log opening transactions he has answered.

4.6.4 Encryptors and hashes tested

By default, the encryption algorithm used is AES, also known as Rijndael. As a hasher, we will use SHA256 as the default option. Both of these algorithms are considered very safe nowadays, and are well tested. Both of them are currently approved by the NSA and should provide strong security for many years. The program can also use other combinations of cipher and hasher algorithms, as long as the encryptor is able to use the keys of the size returned by the hasher. The following combinations have been tested to work correctly together; this list is not extensive and other combinations could work.

- RC2, RC5, RC6, Rijndael,Blowfish and Twofish as encryptors, may be combined with either MD5, tiger or SHA256 hashes.

- RC2, RC5, RC6, and Blowfish may also use the SHA1 hash.

4.6.5 Speed measurements

We have tested the CryptoHasher ciphering program on the same machine used for the system logger benchmarks, a Pentium 3, 866 Mz, 256MB RAM, running Linux 2.4.21. The speed tests have been performed without any system load and with the minimum number of concurrently running programs. The filesystem used was ReiserFS4, and the tests have both been performed under a normal 2.4 Linux kernel and a SELinux enhanced one.

First of all, let's see the impact of selecting different combinations of encryption and hashing algorithms:

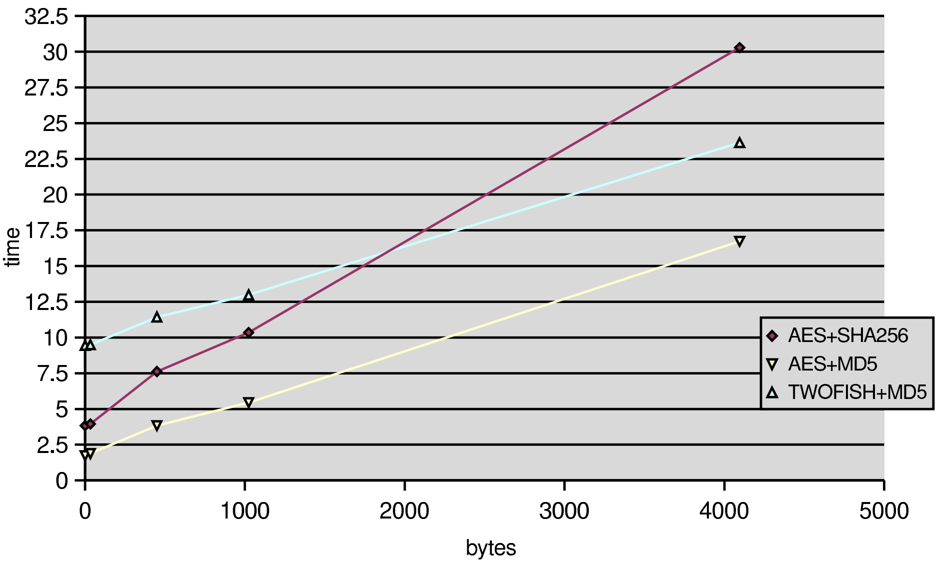
User time needed to secure 50.000 log entries using different algorithm combinations.

Time

Bytes/entry	1	34	450	1024	4096
AES+SHA256	3.84	3.95	7.62	10.35	30.29
AES+MD5	1.73	1.88	3.83	5.45	16.73
TWOFISH+MD5	9.44	9.5	11.43	12.98	23.64

Log entries / second

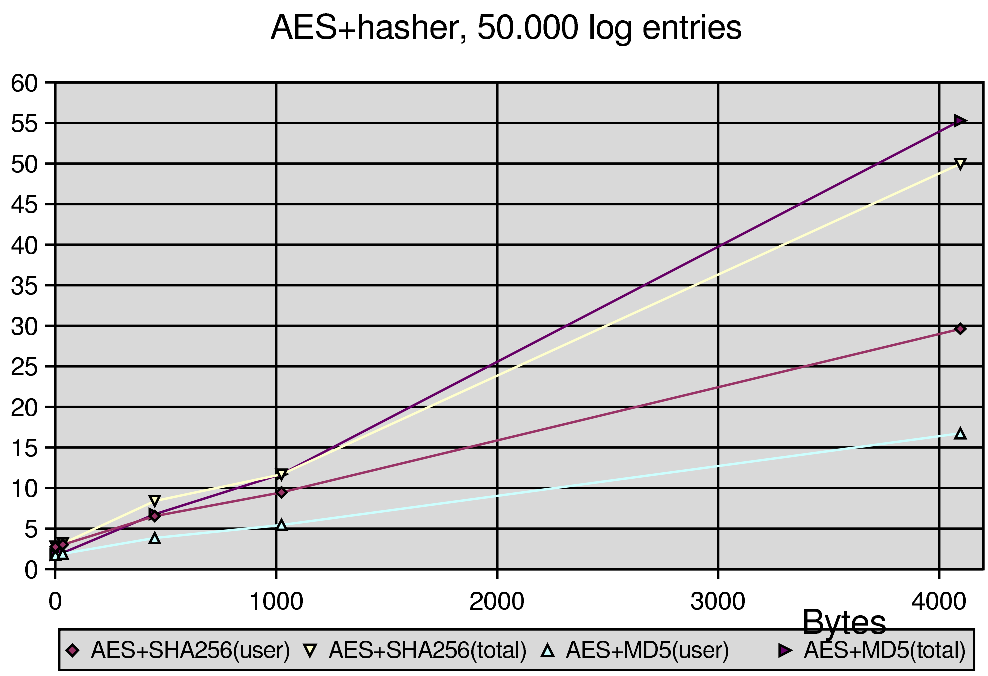
Bytes/entry	1	34	450	1024	4096
AES+SHA256	13021	12658	6562	4831	1651
AES+MD5	28902	26596	13055	9174	2989
TWOFISH+MD5	5297	5263	4374	3852	2115

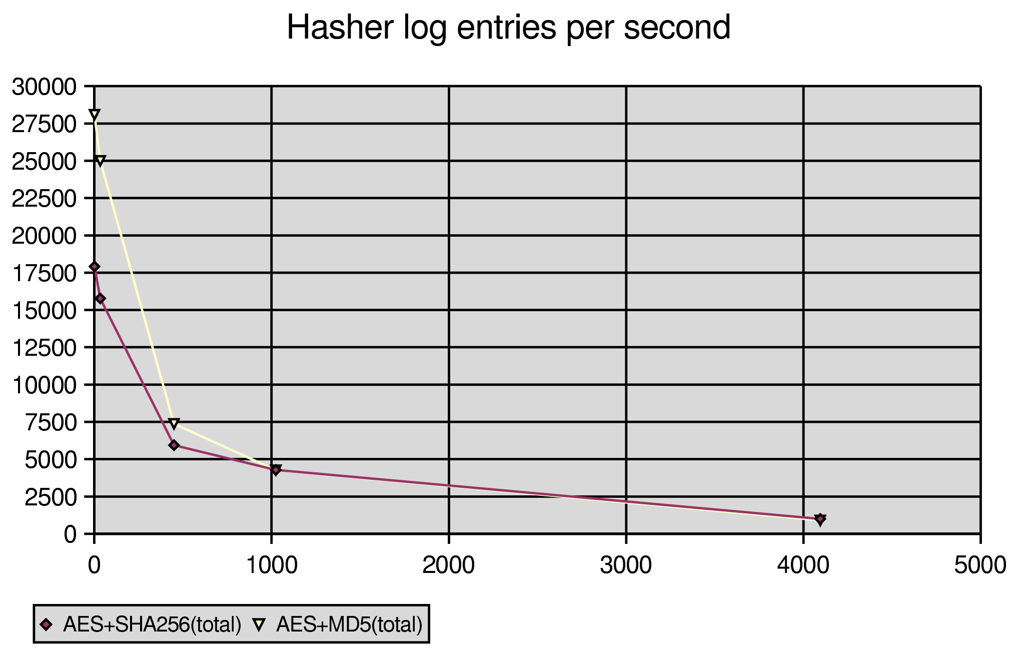


We observe that the fastest of this three combinations is the AES-MD5 one. This is due to the fast MD5 algorithm, which produces hash outputs half the size of the other contender, SHA-256. As MD5 is not believed be very safe nowadays, SHA-256 is preferable for good security.

We can see that the difference between the Twofish-MD5 and AES-MD5 is approximately similar. Both algorithms have similar throughputs, but AES is much faster performing its initial setup. Note that the speed difference between AES-MD5 and AES-SHA256 is not constant, as we would expect due to the fact that the number of hash operations per log entry is fixed and the only variable operation is the encryption. The reason for this is that the while using SHA256, the AES algorithm is using keys 32 bytes long, as opposed to 16 bytes when using MD5. This bigger key makes the cipher slower, but more secure. Anyway, the quick key setup of AES-SHA256 makes it a better option compared to Twofish even while using the larger key size.

We can observe more in detail the effect of the key size and the number of log entries per second on the next figure.


AES+hasher, 50.000 log entries
60
55
50
45
40
35
30
25
20
15
10
5
0
0
1000
2000
3000
4000
Bytes
AES+SHA256(user)
AES+SHA256(total)
AES+MD5(user)
AES+MD5(total)


Hasher log entries per second
30000
27500
25000
22500
20000
17500
15000
12500
10000
7500
5000
2500
0
0
1000
2000
3000
4000
5000
AES+SHA256(total)
AES+MD5(total)

The number of entries processed per second is quite high and should be enough for normal logging operations, specially if we remember that this machine is not very modern. As we process bigger log entries, we can observe that less user time is proportionally used. This happens because the system needs to spend more time writing the data to the hard drive, as he cannot rely on caching mechanisms for the 200MB files used for the last case.

4.7 Integration within SELinux

Because Cryptohasher is intended to be used in environments likely to be hardened against an attack, we have developped rules to use the program within the SELinux security module. The objectives were:

- Restrict the program to perform tasks outside its purpose, in case somebody managed to somehow crack and take control of the program.
- Protect the program configuration and secret files from other processes.

To fullfill these objectives, we have created two SELinux policies: one for the CryptoHasher files, and the other one for the definition of the CryptoHasher domain. Lets begin with the configuration files type enforcement protection:

```
#FILE: /etc/security/selinux/src/policy/file_contexts/\
        program/loghasher.fc
/usr/local/bin/loghasher_server \
        system_u:object_r:loghasher_exec_t
/usr/local/bin/logchecker \
        system_u:object_r:loghasher_exec_t
/usr/local/bin/loghasher \
        system_u:object_r:loghasher_exec_t
/etc/security/loghasher/ \
```

```
          system_u:object_r:loghasher_privfile_t
/etc/security/loghasher/clientA.priv.p15 \
        system_u:object_r:loghasher_privfile_t
/etc/security/loghasher/trustedserver.cert.pem \
        system_u:object_r:loghasher_privfile_t
/etc/security/loghasher/clientA.cert.pem \
        system_u:object_r:loghasher_privfile_t
/etc/security/loghasher/cacert.pem \
        system_u:object_r:loghasher_privfile_t
/etc/security/loghasher/trustedserver.priv.p15 \
        system_u:object_r:loghasher_privfile_t
/var/log/loghasher/ \
        system_u:object_r:loghasher_hashedfile_t
/var/log/loghasher(/.*)? \
         system_u:object_r:loghasher_hashedfile_t
```

As we can see, the files have been given two special new types, *loghasher_exec_t* and *loghasher_privfile_t*. The first one marks the CryptoHasher programs as belonging to the *loghasher_exec_t;* they will be the only programs belonging to this group. The second type, *loghasher_privfile_t*, defines a new type for the configuration and certificate files used by our programs. These files will only be accessible for a program in the *loghasher_t* domain, as we will see in the next configuration file. This ensures that no process, even those owned by root, will be able to read or modify those files, unless they belong to the correct domain. If the legitimate sysadmin wants to set up or modify the settings, he will have to belong to the *sysadm_r* role, as this is the only one apart from the *loghasher_exec_t* which can read the files. But changing to the *sysadm_r* role is a highly restricted procedure in SELinux, and a proper policy will always require entering the root's password to perform this task; the role cannot be granted otherwise. These critical files are thus protected from intentional or unintentional modification. Finally, all the output files from the loghasher program will end in the */var/log/loghasher* directory, with a *loghasher_hashedfile_t* type, and will be configured so that they are not accessible for processes different from loghasher or the system administrator role.

The other policy configuration is the domain definition and restrictions for the CryptoHasher programs. Here they are, explained:

```
#loghasher
type loghasher_t, domain;
type loghasher_privfile_t, file_type;
type loghasher_hashedfile_t file_type;
type loghasher_exec_t, file_type, sysadmfile, exec_type;
```

These three entries define the main skeleton of the loghasher programs. First of all, we define *loghashser_t* as a domain. Using the same syntax as the example SELinux configuration, we define *loghasher_privfile_t* and *loghasher_hashedfile_t* as files, and *loghasher_exec_t* as another kind of file, which also belongs to the system administrator files, and can be executed. These definitions are applied to the files via the previously mentioned *loghasher.fc* file.

```
role system_r types loghasher_t;
role sysadm_r types loghasher_t;
```

Here we enumerate the possibles roles which can run this new domain type. Only programs belonging to the *system_r* and *sysadm_r* roles will be able to run our program, independently of the system's uid.

```
allow loghasher_t loghasher_privfile_t:file r_file_perms;
allow loghasher_t loghasher_hashedfile_t:dir rw_dir_perms;
allow loghasher_t loghasher_hashedfile_t:file {create_file_perms};
allow loghasher_t var_log_t:dir r_dir_perms;
allow loghasher_t var_log_t:file read;
```

We then proceed to allow our program to read, but not modify, its own configuration files and private keys. Also, we allow it to read the files in the */var/log/* dir, in case we want to use it to hash existing log files. Our program is also able to create new files in the */var/log/loghasher* directory. As we have configured *loghasher_hashedfile_t* as a new and unique file type, no other domain at all has access over it, not even the *sysadm_r* role. To effectively read, modify or delete these files, we would need to change their type or load a new policy which allows modification of the files. This is not really an increased security, as these steps can of course be performed from the *sysadm_r* role; this has only been done for the purpose of showing the flexibility of the SELinux policies.

```
neverallow loghasher_t ~{loghasher_exec_t ld_so_t }:file \
     {execute_no_trans entrypoint};
neverallow loghasher_t ~{loghasher_hashedfile_t }:file write;
uses_shlib(loghasher_t);
```

Here we are denying the loghasher program the ability to perform different tasks. First of all, programs in *loghasher_t* will not be able to enter the execution point of files which are not of its own domain. Note the *ld_so_t* exception as we want it to load the shared *lc.so* (cryptlib) library, which whe allow with the *uses_shlib()* macro. The *execute_no_trans* parameter means it cannot enter that domain without transitioning to another domain, we will see why in the following rules. Also note that it cannot write to any file in the whole system apart from the *loghashser_hashedfile_t* file types. This rule is safe, the policy compiler would give a warning and refuse to compile in case some other rule entered in conflict with this one.

```
domain_auto_trans(syslogd_t, loghasher_exec_t, loghasher_t)
domain_auto_trans(sysadm_t, loghasher_exec_t, loghasher_t)
domain_auto_trans(loghasher_t, bin_t, user_t)
```

These rules determine the domain transitions which are possible to and from the *loghasher_t* domain. The first two state that programs which originate from the *syslogd_t* and the *sysadm_t* domain, when executing *loghasher_exec_t* programs, will automatically be tranferred to the *loghasher_t* domain. The final one forces a domain transition to the *user_t* domain whenever a program in the *loghasher_t* domain executes a *bin_t* type of file. But why is this neccessary? In order to gather random information, which will be used for random key generation during the loghasher startup, the Cryptlib libary calls some programs in /bin and /usr/bin, like vmstat, to obtain information which is very variable from system to system. The information gathered from these programs is very difficult to predict, and thus is a good initializer for the library's internal pseudo-random number generator. The random number generator quality is very important, as it is used to generate the shared symmetric key during the interaction with the trusted server, and to create the ramdom initial A_0 and IV. Also, programs on the *user_t* domain are very restricted on their operations. Any user, even root, under that domain[8], cannot delete user files, alter system logs, write to /etc/ files, read or modify protected files, or upgrade his role directly to the sysadm domain[9].

```
allow loghasher_t random_device_t:chr_file r_file_perms;
allow loghasher_t self:capability {ipc_lock};
allow loghasher_t privfd:fd use;
allow loghasher_t sysadm_devpts_t:chr_file { read write };
```

These are some normal capabilities the library needs. First of all, we need to read the Linux random devices, as they will be our main (but not only, as we just explained) source of randomness. We then allow it to use its private file descriptors and process locking, needed to keep the Cryptlib library multithread safe. And in the last rule we allow it to interact with the *sysadm* console (pts device), in case we want to run the program manually and see the error messages on the console.

```
general_domain_access(loghasher_t);
base_file_read_access(loghasher_t);
lock_domain(loghasher_t);
application_domain(loghasher_t);
```

Some other macros needed for normal program functioning, these allow some file read permissions and common safe program operations.

[8] Assuming the default original configuration from the SELinux team is being used, or a similar one.

[9] he can, however, change first to *staff_r* and then to *sysadm_r*, but he will have to enter his password twice.

```
allow loghasher_t resolv_conf_t:file \
    { getattr read };
can_network(loghasher_t);
```

Finally, we allow our program to use the network, and read access to the *resolv.conf* file used for DNS client configuration.

As we can see from the configuration file, the level of detail in which we can describe our program is very high. As we confine our program to work within a restricted environment, the consequences of it being compromised are greatly reduced, as unexpected events are forbidden by the TE/RBAC policy.

4.7.1 Integration with the system logger

We finally tested the loghasher program together with the system logger. For this purpose, we made *metalog* output his data into a pipe, which was used as the input of our program. Here are the results, using the default have tested the default encryption and hashing algorithms.

First, 50.000 log entries of 80 characters:

	Without hasher		With hasher	
	total time	user time	total time	user time
Metalog	2.82	0.36	5.94	0.35
Hasher			“	3.16
Log entries / second	17730		8417	

Our system performed 2.11 times slower while using the log hasher.
Then a different scenario, 1.000.000 entries of 6 characters:

	Without hasher		With hasher	
	total time	user time	total time	user time
Metalog	51.72	6.38	102.07	6.05
Hasher			“	59.29
Log entries / second	19334		9797	

Similar results, our system performed 1.97 times slower than whitout the log hasher. The final performance penality of using the CryptoHasher suite programs is around 200% for our test machine. This speed is much better than the one obtained using PEO or L-PEO. Due to the higher number of operations this should not be the case, we believe that msyslog used a slow library for its cryptographic routines.

While a 2x penality is not really big, we must take this number with caution, as it is obtained whitout any load on the system. Servers protecting their logs will probably have a background cpu usage, and would get worse results. Nevertheless, the performance impact is not as high as it could have been expected for so many operations, and we think the CryptoHasher programs could be tested in working servers.

CHAPTER 5

Hardware solutions

5.1 Introduction

The software-only solutions studied so far, for log authenticating, have the advantage of being cheaply deployed and more or less free, but may also have drawbacks. There is the possibility of flaws in the software or in the machine hosting it, which could ruin our purpose with or without an external attacker. The software may seem safe, but hidden vulnerabilities are often found, as we can see from the numerous security advisories released every month. Once a system is compromised, assuring that previous logs cannot be deleted proves to be difficult without introducing strong kernel modifications, like the ones mentioned in chapter 2; and even these modifications may not be safe enough.

The question is: what hardware features are necessary to assure absolute log safety?. A hardware device that would make submitted information non-deleteable, and if possible its information should be only readable by legitimate system administrators. Also, some degree of feedback mechanism is wanted, so user processes writing to the device know that their log data has been successfully stored, or at least that the device is still working. Thus, hardware solutions for log recording should at least take the role of the Trusted Server in chapter 4, providing a safe locking point for stored data. On the other hand, the hardware device could record all the data themselves, assuming it shares a high bandwith interface with the untrusted client and high storage capacity.

Even with a theoretical perfect storage device available, we must still remember that once the system is breached, we cannot absolutely trust the new logged data anymore, even with the presence of such a device. But implementing a complex authentication protocol with the hardware, along with a finely security-tuned software, will make the odds of an undetected intrusion extremely scarce.

It is difficult to find devices specifically dedicated for logging available in the market. There are, among others, multiple dedicated storage options available (some even with built-in conventional symmetric encryption), as well as encryption-powered network cards, usb or card key-like devices, and hardware encryption add-on cards to free cpu cycles. But none of them assume that the master system could be breached. An encrypted hard drive helps in case of physical theft, but how useful is it if an attacker can read and modify it remotely after breaching into the machine from the internet?

Anyway, even if we have not found a specific device for this purpose, some possibilities do exist and may help in keeping the system logs safe, they are mentioned below.

5.2 Printer

A simple printer can be used as a write only device as it is impossible to remotely modify anything sent to it. But it is a very flawed example, its main problem is that it becomes highly impractical to use for anything but little quantities of data. Also, legitimate access to the actual system logs turns out to be impossible unless physically present where the printer is, and only in paper form.

Nevertheless, a printer may be used for limited sensitive storage. For example, it could assume the role of a trusted server, in the Cryptohasher scenario. In this scenario, the trusted server is in fact providing a safe and authenticated storage of a limited amount of data, the A_0keys. If we assume the physical path to the printer is safe and that we get some feedback from the printer once we have sent something to it, then we are facing a simplified version of that approach that could be used in some circumstances. In this approach, a legitimate user would manually copy the data from the printer whenever necessary; the number of A_0s needed to be hand-copied would be the same as the number of different logs opened by the untrusted server. In fact, it wouldn't even be necessary to hand-copy them, they could all be encrypted with some public key whose private part is not stored on the same machine and at the same time printed. After that, the only required manual activity would be periodically decrypting those keys and checking that they correpond to the printed ones.

5.3 CD-R and CD-RW. UDF.

As part of the thesis work, we have tried writing directly to CD as a way to store data directly to a write only media. Unfortunately, there are some serious drawbacks that do not allow an optimal usage of this medium. The normal data CD-Rom filesystem is the ISO-9660, this filesystem needs to know the files that are going to be stored in it before writing the initial Table of Contents. It is only suitable to store already closed logs and not freshly open ones. Incremental storage of files can be performed, but is impractical if we want to store many small files in one session each, every new session added to the CD adds several megabytes of unused overhead and the CD-Rom gets filled very quickly. Thus, using standard CD-R as storage is only useful to store big amounts of previously logged data. This does not solve our objectives, as an attacker would surely have enough time to modify those logs before they are commited to the CD-R. Nevertheless, if used along with the CryptoHasher programs, secured logs could benefit from this easy and cheap media for medium or long-term storage.

An alternative is to use the UDF filesystem, which is also mature and uses packet-writing to store information on a CD. But the tools to work with this filesystem only work on CD-RW discs and using CD-RWs would not garantee the data integrity. Despite not having found tools to work with UDF on standard write-once read-many CD-Rs, we have nonetheless tested this possibility under Linux and have found that the current tools and kernel support to write to UDF is unfinished!!!!. After that, we tried to use the CryptoHasher suite under a Cygwin-WindowsXP hybrid system with the CD-RW mounted as a standard writeable directory. These are the results:

Entries	100000
Total input (Mb)	18.18
Total output (MB)	26.39
Log entries/s	2911
CPU use	20%
Input throughput	542 KB/s
Output throghput	768 KB/s

As we can see, the limiting factor is clearly the drive's recording speed. It is interesting to note the slower speeds of the initial round, this is beacuse the drive's relatively slow spin-up reaction time adds several seconds of delay to the actual writing. This cannot be avoided unless we keep the drive spinning all the time, but keeping the drive working all the time is probably not safe as the motor has surely not been designed for that and due to the heat that would be transferred to the CD-RW for longs durations, which would negatively affect its reliability.

Thus, using CD-RW or DVD+-R discs for long-term log storage is a cheap and interesting possibility, as some modern CD and DVD recorder drives are recently incorporating the Mount Rainer, a special pure-hardware modification that can work on top of the UDF filesystem. Among other features, Mount Rainer adds automatic defect management to the optical discs, allowing safer data in a non-intrusive and transparent way for the software using it, which solves the unreliability problem of current optical drives.

5.4 Tape and other, specialized hardware

The final alternative, and the more expensive one, relies in using some specialized hardware such as tape recorders. These can vary from cheap tape drivers to medium-price ones and all the way up to high level and specialized networked tape library storage servers, all depending on the customer needs. Unless we find some way to physically force a tape to be written only once, using a local tape is not safer than using a CD-RW, even though they offer a safer medium for long-term storage than current CD-RWs. Thus, systems using tape storage would benefit from the extra security the root power limiting solutions mentioned in chapter 2 can offer.

Other kind of available storages include solid state disks, which at the time of this writing can hold up to two gigabytes of data. These systems act as normal hard drives for the operating system and can be formatted with any filesystem, thus allowing the usage of the special file attributes mentioned in the System Lockdown chapter. A cheap USB 1.1 solid state disk using the ext3 filesystem has been tried, with sustained speeds of 740Kbtyes/second. This is normally enough for log storing and makes these economic devices[1] a viable solution for removable local storage.

[1] which are slow compared to the new USB 2.0 ones, which offer much greater speed

CHAPTER 6

Future work

The CryptoHasher suite is still being developped, and will be available at the sourceforge open source development web site.[1] Having implemented a client-server model for the CryptoHasher suite, future work could study the use of a distributed network of unsecure peers, mutually authenticating one another. Also, on [Sch99], a different approach for key generation is discussed, and could be implemented using the actual CryptoHasher source code.

As the CryptoHasher protocol only needs a very limited amount of safe undeletable storage to work, investigation of devices able to securely store such data could be initiated. Such devices could be integrated in a smart card or some other form of protected solid state memory.

Another possibility would be creating a full hardware implementation of the protocol, freeing the system processor to perform other tasks. A dedicated FPGA chip could process the data as an intermediate step before storage, or could even be directly connected to some external storage facility.

[1] Sourceforge: `http://sourceforge.net`

Chapter 7

Conclusions

In this thesis we have evaluated different approaches for keeping the log files integrity and confidentiality. As they record the system's history, logs are the main location where intrusion or fraud evidence is to be found, and are often the only way to trace back an attack origin.

We have discussed the normal Linux security model, and found it unable to properly secure its own log data. Different new models exist, which offer greatly improved privilege separation. Still rarely used, we believe it is only a question of time before they become widely adopted by the software administrator community, because their deployment increase the odds of keeping data reliable.

Next we have demonstrated that, by using cryptographic tools, confidentiality of data stored before a successful attack can be assured. Along with confidentiality, data integrity can also be proved, only needing a single initial interaction with a secure server, or a small quantity of non delete-able storage; any modification of the data will be detected in the next interaction between the compromised machine and the server. As we know if the log data we have gathered from our machines is trustworthy, it is usable by us as cybercrime proof, while at the same time inutile for the infiltrator.

The performance impact of these tools is not negligible, but the increased security they offer is worthwhile. Security modules are being developed and improved, while the overhead involved with the cryptographic operations is fixed. Thus, with the current computer industry still following Moore's Law, we believe it will not be long before the cost of these operations is not relevant anymore, and computers begin using secure kernels and cryptography on a wider basis. This will ultimately make the Internet and its computers safer while we enter the digital era.

Bibliography

[Sch97] B. Schneier and J. Kelsey, *Cryptographic Support for Secure Logs on Untrusted Machines.*The Seventh USENIX Security Symposium Proceedings, USENIX Press, Jan 1998, pages. 53–62.
Also appearing at: ACM Transactions on Information and System Security, vol 2, No. 2. May 1999, pages 159-176.
Available online at :`http://www.counterpane.com/secure-logs.html`

[Bau02] Michael D. Bauer, *Building Secure Servers with Linux.* O'Reilly, October 2002. ISBN 0-596-00217-3.

[Sch99] B. Schneier and J. Kelsey, *Minimizing Bandwidth for Remote Access to Cryptographically Protected Audit Logs.* Second International Workshop on the Recent Advances in Intrusion Detection (RAID '99), September 1999
Available online at `http://www.counterpane.com/auditlog2.html`

[Lpeo] E. Kargieman and A. Futoransky. *VCR and PEO revised.* Core SDI S.A. papers, October 1998. `http://www1.corest.com/files/files/11/PEO.pdf`

[Los01] P. Loscocco and S. Smalley.*Integrating Flexible Support for Security Policies into the Linux Operating System.* Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01), June 2001

[Los01] P. Loscocco and S. Smalley.*Meeting Critical Security Objectives with Security-Enhanced Linux*. Proceedings of the 2001 Ottawa Linux Symposium.

[Sma02] S. Smalley, C. Vance et al: *Implementing SELinux as a Linux Security Module.* May 2002.`http://www.nsa.gov/selinux/module-abs.html`

[Sma03] S. Smalley. *Configuring the SELinux Policy.* January 2003.`http://www.nsa.gov/selinux/policy2-abs.html`

[SELinux] SELinux documentation at `http://www.nsa.gov/selinux/docs.html`

[GrSec] GrSecurity documentation at `http://www.grsecurity.net/papers.php`

[Honey] Project Honeynet: `http://project.honeynet.org/`

[LinSec] Linux Security: `http://www.linuxsecurity.com/`