USING INDEPENDENT AUDITORS FOR INTRUSION DETECTION SYSTEMS

by

Jesus Molina

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Sciences

University of Maryland, 2001

Approved by _____

Chairperson of Supervisory Committee

_____

Program Authorized

to Offer Degree _____

Date _____

University of Maryland

Abstract

USING EMBEDDED AUDITORS FOR INTRUSION DETECTION SYSTEMS

by JESUS MOLINA, Master of Science, 2001

Chairperson of the Supervisory Committee: Professor Virgil Gligor

Department of Electrical Engineering

A basic cornerstone of security is to verify the integrity of fundamental data stored in the system. This integrity checking is being achieved using integrity tools such Tripwire, which depend on the integrity and proper operation of the operating system, i.e. these applications assume that the operating system always operates correctly. When this assumption is not valid, the integrity applications cannot provide a reliable result, and consequently may provide a false negative. Once the operating system is compromised, a novice attacker, using tools widely available on the Internet (rootshell.com, etc), could easily defeat integrity tools that rely on the operating system.

A novel way to overcome this traditional integrity problem is to use an independent auditor. The independent auditor uses an out-of-band verification process that does not depend on the underlying operating system. The

resultant system provides extremely strong integrity guarantees, detecting modifications to approved objects as well as detecting the existence of unapproved and thus unsigned objects. This is accomplished without any modifications to the host operating system.

As an auditor we use a StrongARM EBSA-285 Evaluation Board, with a SA-110 microprocessor and 21285 core logic.

ACKNOWLEDGMENTS

The author wishes to thanks professor William Arbaugh, for the technical and financial support, but specially for trusting me. I am extremely grateful to my advisor professor Virgil Gligor for his continuous feedback and kindness even when I did not deserve it, and to professor Charles Silio for agreeing to be part of the committee despite having an extremely tight schedule.

I also wish to thank Jessica Peterson, for being who she is and for correcting my English.

Finally, thanks to all the people that helped me along the way, which include A. Mishra, the people on the ARM-linux mailing list, R. King, M. Van Doesburg and the people of the MISSL lab, especially A. Agnew

TABLE OF CONTENTS

LIST OF FIGURES

C h a p t e r 1

1 INTRODUCTION

Computer Systems have been made increasingly secure over the past decades. However new attacks and the spread of harmful viruses have shown that better method must be used. One approach gaining increasing popularity in the computer community is to use Intrusion Detection Systems (IDS) to improve the security of the systems.

Intrusion Detection Systems detect an intruder breaking into your system, or a user performing any illegitimate action or misuse of system resources. Using a common analogy, having an Intrusion Detection System is like having a “burglar Alarm” in your house. The alarm will not prevent the burglar from breaking into your house, but it will detect and warn you of the problem.

Since the publication of the first research in intrusion detection systems, a large number of applications have been developed, using different techniques. One technique to accomplish this detection is the use of file system integrity checkers. When a system is compromised, an attacker will often alter certain key files to provide continued access and to prevent detection. The changes could target the kernel, libraries, log files or other sensitive files. These file system integrity checkers detect a change and trigger an alert.

Continuing with the “burglar alarm” analogy, this kind of IDS will check if forged copies have replaced your Picasso paintings and make sure your jewelry is still there.

To ensure the integrity of the file system, two approaches can be followed.

The first approach is to create a secure database, which is usually composed of hashes of the important files. The stored hash will be periodically checked against the new computed hash. A hash-function is a one-way transformation that mathematically computes a digest of the message, transforming a message from an arbitrary length to a message of fixed length. This method is used with tools such as Tripwire [13], Aide [19], Dragon Squire [20] and others.

The second, more recent approach is to create digital signatures of sensitive data, such as executable files, using public keys. Using asymmetric cryptography, it is possible to sign files. Briefly, asymmetric cryptography requires two keys, one for encryption and one for decryption. You can encrypt the binaries you want to release with your own (private) key, and post the other (public) key in a secure database. If these files are to be opened, the OS will decrypt the file using the public key and ensure that the file has not been corrupted. In practice, the binary itself is not encrypted. Rather, a hash of the binary is, then this hash is appended. This approach is discussed in [16], and [17].

Both approaches have advantages and drawbacks that will be discuss in depth in the following chapters, but they also have a common flaw. The auditing relies on the Operating System. All the previous applications have made the assumption that the OS itself is not corrupted. Once the operating system is compromised, a novice attacker, using tools widely available in the Internet, could easily defeat integrity tools that rely on the operating system. For example, in the Linux operating system, redirecting system calls using kernel modules will compromise the system [11]. Usually a system is trusted because a good IDS has been running for a long time. However, the system typically has not been checked before the installation. Hence the IDS may be running in an insecure environment.

Finally, most proprietary operating systems are trusted by default, assuming that the OS will work as expected.

Another important flaw in the database approach is the large time intervals between checks in the database, which is a consequence of the large computational requirements involved. For example, an attacker with knowledge of the checking schedule may break into the system between these checking points and change the operating system or the IDS software itself, leaving the system in an insecure state.

This work develops a novel way to overcome the problems of a traditional Integrity Checking Systems. Our approach is to use an independent auditor, i.e. a completely standalone and independent device to perform the integrity detection checking.

More specifically, this work will be focused on implementing an integrity checking system using an ARM microcomputer plugged into the PCI bus. The auditor will run an Open Source version of Linux as well as an open source BIOS. This auditor uses the approaches previously discussed to check the integrity of the system, completely independent of both the host machine and the OS running in the host machine.

In both approaches the auditor provides the system an “in depth” defense. This concept means that an all-powerful attacker (i.e. a supervisor with access to the machine, knowing every security device, program and password) with only the restriction of physical access to the machine will not be able to bypass the integrity checking system.

Using the first approach, the device will create a secure database of computed hashes, which will be periodically compared against the new computed hashes in the machine. An auditor devoted to this task could do the checking more often than normal machine could. This checking will be at random times with a certain mean and variance. At every check

time, after the comparison is done, the auditor will keep a copy in non-volatile storage of important log-files and data. These log-files cannot be secured using this approach, as they change frequently. But keeping a copy of them in a known secure state will allow us to know, in the event of an alert, information about the state of the machine before the attack, creating a “black box” of your computer. A discussion of the importance of secure audit logs can be found in [10].

In the second approach, the public keys to perform the integrity checking are stored in the auditor. The auditor could make active requests to verify signed files, or act passively by "sniffing" the bus and recreating the file-system in parallel, avoiding execution of signed files in real time. In this approach, not only is the corruption of the system prevented, but only files from trusted parties will be allowed in the machine.

Although this work is based on implementing an integrity checking system using an out-of-band approach, inside the auditor other IDS can be implemented. As an example, the auditor could sniff the traffic passing by the Ethernet card and alert the machine of possible network attacks.

In the next sections some terms used throughout this document are described.

1.1 Embedded Systems

Personal computers are widely used. They are usually composed of a hard disk, some kind of video device (such a monitor), and other hardware and Input/Output devices. These computers are designed to perform a variety of tasks.

However, there are other devices, called embedded systems, which are also composed of hardware and software, but are designed to accomplish a specific function. For example, an

embedded system, composed by a microprocessor and a LCD screen could be used in a plane for sending, receiving and processing positioning signals from the control tower. The software and the hardware in that device are devoted only to this purpose. Examples of embedded systems applications would be VCR, alarm clocks and video-gaming devices.

The function of embedded systems can usually be imitated by replacing the embedded system with an equivalent device completely built in hardware, without any kind of processor or software. However, embedded system using multi-purpose processors are easier to build, and are cheaper. In fact, as the availability of cheaper tiny processors have risen in the market, the number of embedded systems has also risen. As an example, PDAs are a new application of embedded system, which is having a great impact in the market.

Software is usually designed to be operated by personal computers. A piece of software, designed for one personal computer, will also run in any other personal computer if they share the same architecture and OS. In embedded systems, software is created for a specific platform and hardware, and is unlikely to work in any other platform. In fact, the software in an embedded application is usually highly specific to the hardware to keep low cost.

There are several key hardware components common to most embedded systems.

The main component is the microprocessor. Typical target processors are Intel x86, Motoral 68k, MIPS R3xxx, R4xx0, Intel strongARM, etc. In order to run software, the embedded system should have some kind of storage. It has some temporary cheap volatile storage (i.e. RAM) and some non-volatile storage to keep the software and data (i.e. flash storage)

Software for embedded systems is usually built using development tools in a general-purpose computer, such as an Intel x86 running Windows™. The created software is then

"translated" to the embedded architecture using a cross-compiler, a tool that compiles software in one architecture for use in other architectures.

Usually, the term Host refers to the computer, in which the software is developed, while the term Target is the embedded system in which the software is ran. Details of the Target, Host and the cross-compiler used to build the embedded coprocessor can be found in chapter 4, Designing the System. A list of common targets and hosts can be found in fig.1.1

Figure 1.1 Common Host platforms and Target Processors

Host Platforms	**Target Processors**
DEC Alpha Digital Unix	AMD/Intel x86 (32-bit only)
HP 9000/700 HP-UX	Fujitsu SPARClite
IBM Power PC AIX	Hitachi H8/300, H8300H, H8/S
IBM RS6000 AIX	Hitachi SH
SGI IRIS IRIX	IBM/Motorola PowerPC
Sun SPARC Solaris	Intel i960
X86 Windows95/98/NT	MIPS R3xxx, R4xx0
X86 Red Hat Linux	Mitsubishi d10V, M32R/D
	Motorola 68k
	Sun SPARC, MicroSPARC
	Toshiba TX39

Although software for embedded system was monolithic in the past, new, more complex embedded systems with higher needs, will benefit from the inclusion of an Operating System. These embedded operating systems will be discussed in the next section, along with the ARM architecture (used by the embedded coprocessor of this work) and some aspects of the memory devices.

The Acorn's ARM architecture is a RISC load/store architecture designed to provide very high code density suitable for embedded applications, which require low power consumption. The base architecture does not provide any hardware support for floating-point arithmetic. However, it could be implemented using software emulators of the floating-point unit.

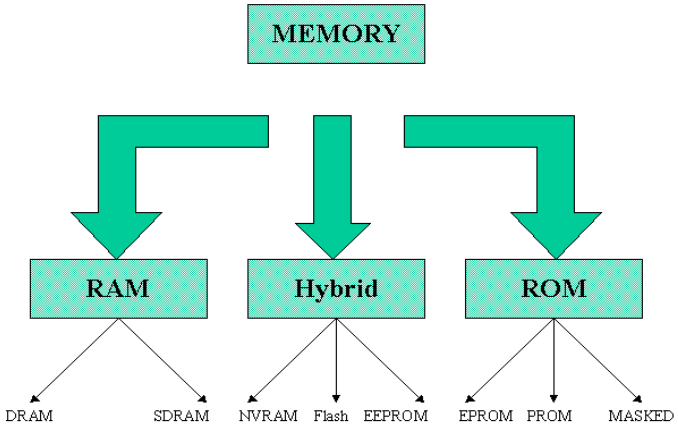
The first ARM chip was released on April 26 1985, making ARM the first RISC commercial processor. Since then, there have been different revisions, creating four different version of the chip. The last, ARM v4 is the one used by our platform. More information about the ARM architecture can be found in [3].

1.1.1 Memory

Different types of memory are available for the use in embedded systems. Memory devices are usually divided into two main types: RAM (Random-access) and ROM (read-only). This is a very rough division, however, as there are subtypes of each and hybrids of both memory types. RAM devices can be used to read and write, but the storage is volatile. If there is no power, the data stored in the RAM will disappear. In a ROM device, the data stored at each memory location can be read, but not written using software. It is possible, however to erase and rewrite data using hardware.

See figure 1.2 for an overview of the different types of memory and subdivisions.

Figure 1.2 Common memory types in embedded systems



The Embedded processor used for the project utilizes Flash memory as the non-volatile storage. Flash memories are hybrids between ROM and RAM devices. There are low cost, non-volatile, and fast to read but very slow to write. Nevertheless, it is possible to write to them using software. A file-system can even be created using flash-memory. These advantages actually make flash-memories the most popular type of non-volatile memory for embedded systems.

1.2 Hard disks

In most computers the hard disk is the principal mass storage system. It stores the data files and programs, and delivers the data to the CPU when it is needed. Hard disks usually differ in diverse parameters from each other, from technology to speed. However, they share some common ground. Hard disks are rigid platters, composed of a substrate and a magnetic medium. The base must be non-magnetic. Both sides of each platter are coated

with a magnetic medium called a thin-film medium. This stores data in magnetic patterns, with each platter capable of storing a billion or so bits per square inch (bpsi) of platter surface. This data is organized into larger "chunks" tipically to allow for easier and faster access to information. Each platter has two heads, one on the top of the platter and one on the bottom, so a hard disk with three platters (normally) has six surfaces and six total heads. Each platter has its information recorded in concentric circles called tracks. Each track is further broken down into smaller pieces called sectors, each of which holds 512 bytes of information. Nowadays, hard disks are not just dumb devices to store information. Embedded in the device is usually an intelligent controller, which handles the requests from the operating system. For example, the IDE interface is one type of controller. A further explanation of this specific type of controller can be found in chapter 5. The operating system uses a File System as a way to organize files and directories on the hard disk. The File-system is dependant on the operating system. Linux, for example, uses the ext2 file-system by default, while the desktop version of Windows uses the FAT-32 file-system.

1.3 PCI bus

The PCI (Peripheral component Interconnect) is a complete set of specifications defining how different peripheral should interact. A implementation of these specifications is the PCI bus. The PCI Local Bus is a high performance bus for interconnecting chips, expansion boards, and processor/memory subsystems. Devices attached to the PCI bus (PCI devices) are automatically configured at boot time by the system. A bus number, a device number and a function number identify each PCI device. Each system could have

up to 256 buses; each bus up to 32 devices and each device can handle up to eight different functions. Usually systems have at least two PCI buses. These buses are connected to each other using bridges, a special type of interface designed for this task. The PCI devices share address space, permitting communication between each other. Every device also has specific configuration registers that can be accessed by the system to configure or read information about the device.

1.4 Operating Systems

Any machine could have a huge monolithic program that completes all the tasks of the systems. While this approach is sometime used in embedded systems, it is usually not feasible for more demanding systems where it is desired to complete several tasks or continually develop new software. An Operating System is a program that completes several objectives, including:

- Provides Abstractions: Hardware has low-level physical resources with complicated, idiosyncratic interfaces. An Operating System provides abstractions, which present clean interfaces. The objective is to make the computer easier to program and provide better communication with hardware devices.
- Provides Standard Interfaces: This allows portability between different systems.
- Mediates Resource Usage: This allows multiple processes to share resources fairly, efficiently, safely and securely. A common example is the capability of processes to share one processor, allowing the possibility of more than one task running in the same processor at the same time.

The OS is composed of different programs. The core of the Operating System is called the Kernel. When the system boots, the kernel is usually loaded into memory from a non-volatile storage device. As the other programs of the Operating system are not crucial to its work, in this document the terms OS and kernel will be used interchangeably.

There are essential differences between Operating Systems and embedded operating systems. Current Operating systems consist of huge pieces of code, devoted mainly to simplify the usage of the computer making it more attractive to the common user. In the other hand, embedded operating systems are not user oriented, but committed to a precise work. Embedded Operating systems are typically small, as they do not require user interfaces, or the need to communicate with certain hardware devices, such monitors and keyboards. Furthermore, the storage devices are usually reduced, so the OS size is strongly limited by the lack of storage.

There are several Operating Systems for embedded systems. Some examples are ecos [21], Linux [22], windows CE [23] and many others. Actually, anyone could create his own OS. After all, an Operating system is just another piece of software.

The system in this study uses a port of the Operating System Linux for Arm processors. Chapter 4 will discuss why embedded Linux was the chosen Operating system.

1.4.1 Linux Operating System

Linus Torvalds, a Finish student of computer science, created Linux in 1991. It was first developed as an Operating System for IBM compatible personal computers based on an Intel 80386. But there has been subsequent development on this OS. As the source code

was placed in the public domain, freely available, a group of LINUX activists formed to develop the operating system. This movement is growing daily a more reliable OS.

Linux is based on the well-known commercial Unix operating system. Looking in the documentation from the last Linux release we can read a self-description of Linux. [24]

> *Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX compliance.*
>
> *It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management and TCP/IP networking.*
>
> *It is distributed under the GNU General Public License.*

Linux has some advantages when compared with other operating systems. The first big advantage is that Linux is free. In fact, Linux is distributed under the GPL (General Public License). The GPL not only allow the freely use of Linux, but also allows the user to modify the source code and redistribute the new version, as long as the new code will be distributed under the GPL, this ensures that the new versions of Linux will also be free.

Other benefits of Linux are its high compatibility with other OS and the strong technical support available through the Linux community. An extensive literature is available concerning Linux [6].

Linux attempts to make a clear division between Hardware dependent and Independent source code. Therefore Linux is highly portable and greatly suitable for embedded

applications. From the primary x86 platform Linux has been ported to ARM, alpha, m68k, mips, ppc, sparc, sparc64 and s390.

In the system of this study ARM Linux is used, a successful port of Linux to the ARM processor based machines, which was developed mainly by Russel King with contributions from other sources [25].

1.5 Interrupts. Polled mode vs. Interrupt mode

An interrupt is an event that alters the sequence of instructions executed by a processor. These events correspond to electrical signals generated by hardware circuits. Usually we differentiate between synchronous interrupts, generated by the CPU after terminating an instruction, and asynchronous interrupts, generated by other hardware devices at arbitrary time. The synchronous interrupts are usually called exceptions, while the term interrupt is usually used to refer to asynchronous interrupts. For example, the keyboard will generate an interrupt every keystroke, while the CPU will raise an exception if an abnormal situation occur. The interrupt requests by I/O devices are called IRQ.

Interrupts are used because operations with peripherals are usually unpredictable. These events could be a packet arriving at the system or pressing a key in the keyboard. These events are usually stored in a status register, and the operating register should react when this status register is updated. The two techniques available to monitor I/O (Input/Output) operations are polling mode and interrupt mode.

In polling mode the CPU will check every certain time the status of the status register. This method is simple, but it wastes a huge amount of time, as the CPU should perform the task of verifying the register. Usually, I/O operations are implemented using interrupts. When

an event occurs, the device will signal the CPU to raise an electrical interrupt. The operating system will have a table with a pointer to the code to call when this certain interrupt is called. The action to the event signaled by the interrupt will be handled by this code.

Using a common comparison, the polling mode is comparable to staring to a door with the door open while waiting for a guest. The interrupt mode could be compared to closing the door and waiting for a knock in the door to open it.

1.6 Hashing, Public key Cryptography and digital signatures

At the beginning of the chapter the terms Hash Functions and file signing were introduced. Here they will be further developed.

As previously stated a hash function is a mathematical function that computes a digest of the message, transforming it from an arbitrary length to a message of fixed length. The principal attributes of a secure hashing function are the following:

- It is a one-way process. It is mathematically infeasible to reconstruct the original data from the hashed result.
- Given a hashed message, it should be mathematically infeasible to find a message with the same given digest
- The hashed result is unpredictable. Given a source of data, is extremely difficult to find another set of data sharing the same hash result. This property follows from the two previous properties.

Hashing functions are usually used in computer security to check the integrity of data by computing the hash function of the message and comparing it with a certain known previously hashed digest. Notice that hash function provides only integrity.

The following are some popular hash functions

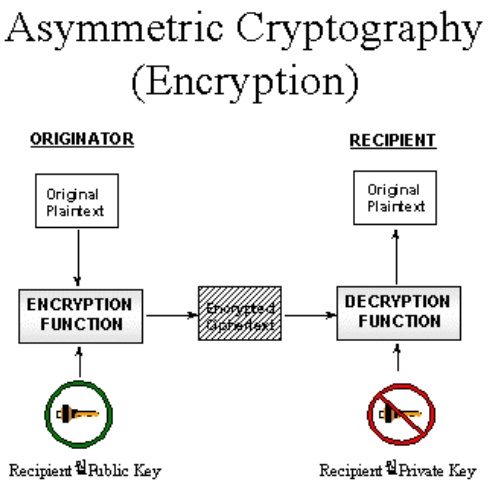
MD2 and MD5, created by Ron Rivest, which produces a message digest of 128 bits

SHA, proposed by the NIST (National Institute of Standards and Technology), producing a message digest of 160 bits.

Digital signatures add authentication and non-repudiation to an integrity checking system. It is possible to check if a file has been signed from a certain party. Therefore, this party cannot claim that they are not the creators of this file. Because encrypting the entire message will be computationally expensive, a digest (hash) of the file is first computed.

File signing involves using a public key cryptosystem. Although there are different public key algorithms, they share one general concept. There are always two components used for operation in the input data. One of the components is the private key, while the other is named public key. If we encrypt the input data using the public key, encrypting the ciphered data again using the private key will retrieve the original data.

Figure 1.3: Asymmetric Cryptography

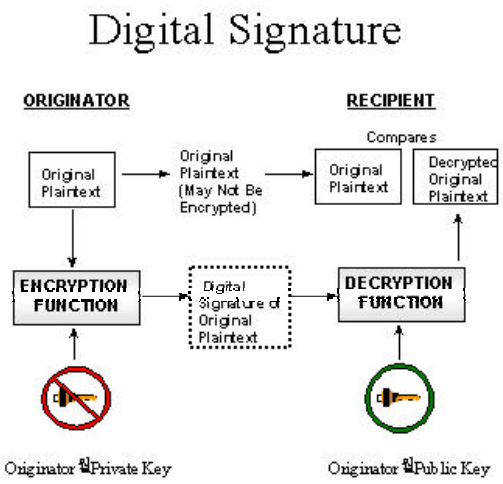


Hence one key can be used to invert the operation performed by the other.

The public key is then distributed and made available in secure databases, which will match the identity of the user with his/her public key.

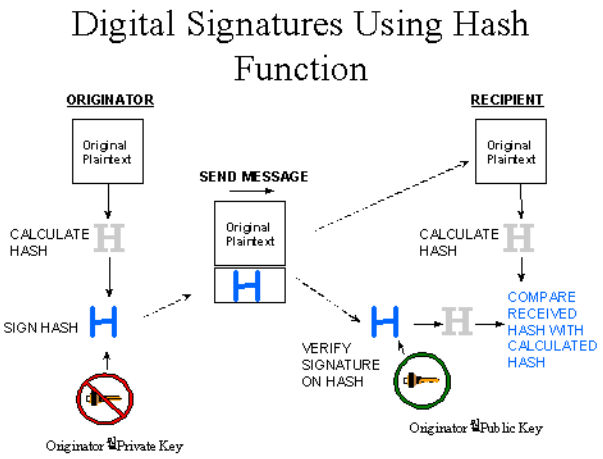
The user then can encrypt a message using his/her own private key. To know if a certain user signed the file, the key can be retrieved from the secure database and used to decrypt the digest. Next the digest of the message will be computed and compared against the decrypt signature. If they are different, either the file has been corrupted or another user has encrypted the digest. If there is no change, then it is certain that that particular user signed the file.

Figure 1.4 Digital Signatures



While public key cryptography is a powerful tool, the operations involved require an extensive use of the processor and are usually slow. Signing a complete message would computationally expensive. Usually, not the entire message is encrypted, but rather the hash of the message is. As the computed hash is always the same size, no matter the size of the plaintext, encrypting using asymmetric operations is much easier.

Figure 1.5: Digital Signatures Using Hashing Functions



1.7 Organization

This work is organized in seven chapters. The first chapter is this introduction, defining some terms that will aid in the comprehension of material in following chapters.

The second chapter is an overview of the intrusion detection systems. Even if the implementation of the system in this project only covers a Integrity Verification System, reviewing the different possibilities and aspects of the Intrusion Detection Systems will give a background for further development, as well for a better understanding of the system.

In the third chapter the implementation of an Out of Band verification system will be explained, which will include details of the system as well as a review of other approaches which use the same system in a theoretical way.

The next chapter is devoted to the technical review of the implemented system, completed in appendix A with the code of the BIOS, applications and Kernel patches.

Chapter number five includes graphics and data on the performance and accuracy of the system, along with a discussion of the results.

In the sixth chapter the overall conclusion of the work will be discussed, and the future work regarding implementing Out of Band intrusion detection systems.

In the final chapter I will include the bibliography used for writing this work.

C h a p t e r 2

2 Overview on Intrusion Detection Systems

Diverse security mechanisms are usually implemented in the system to prevent unauthorized access. However, as new security flaws appear, completely preventing them is usually unfeasible. But once the intrusion has occurred, it is possible to detect it and to minimize the damage to the system using special agents.

Intrusion detections systems (IDS) attempt to detect these intruders breaking into your system or a legitimate user misusing system resources

2.1 Definitions

In this section the common concepts used trough this and the following chapters referring to the intrusion detection systems are described. Most of the definitions are extracted from [28] and [1].

2.1.1 Trust

The technical definition of trust in the computer security field is the degree of confidence in which the system behaves as expected. The different levels of trust match the levels of confidence in the system. Following this definition, a non trusted system is one in which no warranties about its behavior can be stated. The formal concept of trust is complex and often neglected. Usually, system or agents (for example, operating systems) are granted the

"trusted" state without any warranty of their behavior. A more extensive development of the concept of trust can be found in [27]

2.1.2 Threat

A threat is any event with the potential to harm any essential element of the system. These types of threats can be of diverse nature, and countless. Fire in the building where the computer is hosted is as much a threat as computer virus is (and possibly more harmful). Hence we can divide threats into intentional (viruses, hackers) or incidental (fire, floods).

We describe an intrusion or attack as any set of intentional actions that attempts to create a threat.

From the originator point of view we can divide the attacks in:

- External attacks, performed by an intruder foreign to the system
- Internal attacks, performed by authorized users.

Types of penetration include [27]

Espionage, including any disclosure of sensitive data or leakage of information

Denial of Service, including the disruption of any service provided by essential resource elements and the destruction of essential resource elements.

The term penetration is used to refer to a successful intrusion.

2.1.3 Vulnerability

Vulnerabilities are flaws in the system that exposes it to penetration. Usually, in an intrusion, the attacker will try to exploit the vulnerabilities of the system to gain unauthorized privileges. These weaknesses of the system could have their origin in flawed software or hardware. In that case, they are called technical vulnerabilities. However, the

vulnerabilities sometimes occur as a result of a deficient configuration or/and management of the system. These are called management or procedural vulnerabilities.

2.1.4 Security Policy

A security policy documents the producers or models necessary to accomplish security requirements. Security policies can be divided into:

- Procedural security policies, which refer to a document outlining the security goals and the resources to be used in order to achieve these goals, and
- Formal security policy, which consists in a mathematical model containing the states and operations to move from one state to another and the constraints of when and how these operations may exist.

2.2 Importance of Intrusion Detection systems

To avoid penetrations, systems usually base their security in tools such as firewalls, access control mechanisms and others.

But incidents like the Internet Worm [30] or recent events, such the apparition of malicious worms like “I love you” [31] or Code Red [32], have shown that today’s systems are far from being secure against possible penetrations.

It could be argued that better, more secure systems with better tools and stronger cryptography will stop these threats from occurring. However, the complete prevention of intrusion is unrealistic in the present days:

A system completely secure and free of vulnerabilities requires software to be bug free. To accomplish this the systems administrators should revise the code for every program to be

run in the system. This is obviously infeasible and almost impossible for most systems, as several software packages and operating systems do not make their source code available.

Security tools rely on the competence of the user. Even the best crypto-system can be broken if the password has been written on a piece of paper. Also, an insider could abuse his or her privileges. Humans are usually the weakest link in a secure system.

Strict, secure mechanisms reduce the efficiency of the system. For example, longer passwords will delay the access of the user and better, more sophisticated encryption algorithms will make programs slower.

In [17] evidence is shown that even well known vulnerabilities with available patches are exploited for a long time after the patch is released.

Hence, even in extraordinarily secure systems, penetrations could arise. If a penetration has occurred, a secure system should be able to react, detecting the penetration as soon as possible and storing audit data about the penetration, to prevent such an attack in the future and to trace the attacker. IDSs, however, usually do not perform any reactive measures to an attack.

2.3 Goal for an Intrusion Detection System

The Intrusion Detection Systems should try to achieve the following goals:

- It must be difficult to bypass
- It should run continually without human supervision. The system must be reliable enough to allow it to run in the background of the system being observed.
- It must be fault tolerant in the sense that it must be resilient to unexpected fatalities, as systems crashes

- It must resist subversion. The system most be capable to detect malicious corruption of its own code
- It must impose minimal overhead on the system.
- It must be easily tailored to the system in question. Every system has a different usage pattern, and the defense mechanism should adapt easily to these patterns.

The inaccuracy of an Intrusion detection system could be classified in false positives, false negatives and subversion errors.

False positives are legitimate access or actions in the system categorized by the Intrusion Detection system as intrusions. False positives should be minimized, as a large number of false positives obscure the task of separating real attacks from lawful actions. In such a scenario, the maintainer of the system could overlook an attacks, even if it has been detected due to the large amount of detected attacks.

More serious are the false negatives. In this case, an intrusion is not detected, and labeled as a normal behavior of the system. Causes of false negative are diverse, but usually new attacks not catalogued bypass systems based on pattern matching. This issue will be further discussed in the following section.

Subversion errors are similar to false negatives. An intruder with certain knowledge of the intrusion detection system will take advantage of known flaws of the intrusion detection system itself or can change the behavior of the intrusion detection system to allow unlawful action to be labeled as legitimate.

2.4 Types of Intrusion Detection systems

Using the academic model of intrusion detection systems we can divide them in two models

Anomaly detection model: The intrusion detection systems following this model detect intrusions by looking for abnormal activities in the system [33].

Some approaches using this model are:

Statistical approaches where profiles of user behaviors are generated and compared against the actual behavior of the user.

Predictive pattern generation, where the intrusion detection system attempts to predict the future using occurred events, and triggers an alarm if known events occur or if events are matched as an intrusion.

Some examples of IDS using the anomaly detection model are [35] and [39].

The use of this approach usually leads to a large number of false positives but can prevent attacks not previously identified.

Misuse detection model: Intrusion detection systems following this model detect intrusions by matching activities against patterns of known intrusion techniques (signatures) or system vulnerabilities [18].

Approaches using this model include:

- Keystroke monitoring, matching attack patterns using pressed keys.
- State transition analysis, where the system is modeled as state transition diagrams, and certain state are labeled as “SAFE” and other as “UNSAFE” [36].

Approaches using this model are [37] and [38].

The use of this approach leads to high accuracy ratios but is unable to detect novel attacks.

2.5 Integrity Verification Systems

Another, more subtle type of Intrusion Detection System are integrity checkers. Integrity checkers will not detect the intruder even in the case of a successful penetration. The purpose of using integrity checkers as an Intrusion Detection System is to detect change in the files-systems after the penetration. Motives for altering the files-systems are diverse. The intruder could try to cover his or her tracks by removing the logs or place Trojan horses in the system to regain access in the future. Integrity checkers will detect these attempts and trigger alerts.

It could be argued that the use of integrity checkers is not necessary if we use other types of Intrusion Detection Systems, which detect an intruder even if the file-system is not changed during the penetration. However, research performed by DARPA in 1998 showed that even the best IDS had detection rates below 70% [40]. Usually, the undetected intrusions were novel attacks, which could lead to supervisor access to the system. The same research in 1999 showed similar numbers; the detection ratio was still below the 70^{th} percentile [41]. This data shows that intrusion detection systems are likely to fail detecting novel attacks. Using anomaly detection systems may prevent and raise this ratio, but state-of-the-art anomaly detection systems are not suitable for large commercial networks because they provide a large number of false positives.

Integrity file checkers give the system another layer of security, allowing systems to avoid being left in a completely unreliable state after a penetration has been discover.

Integrity file checkers can be divided into two main categories,

2.5.1 Database integrity checkers

Database integrity checkers create a unique identifier for every file to be checked. This identifier, usually some type of hash function, is then stored in a secure database. At certain intervals, the identifier will be recreated and compared against the stored identifier. If the identifiers differ, an alert will be triggered. The saved record usually contains other significant file information such as length, time of last modification and the owner of the file.

The stored identifier should be computationally simple to generate but impractical to reverse. Also, it should be impractical to create a random file with the same identifier as another file. As was discussed in the introduction, these goals can be achieved using one-way-functions, also called hash-functions. The hash functions to be used in the database depend on a balance between performance and security.

Usually, along with the database where the identifiers are stored, a policy file exists where the files to be monitored are declared. The policy file is necessary because not all files in the system can be monitored. Files that change frequently, such as log files and database cannot be checked, as this will lead to numerous false positives. This situation is undesirable as the task of separating the false positives from the real alarms will be complex. Typically the configuration file also gives some parameters, which specify constraints to the file to be monitored. This allows the administrator of the system to check other types of files such as files that only increase or decrease in size, or change contents but not file attributes.

A an example, fig2.1 shows a policy file from Tripwire which is an implementation of a database integrity file checker

Figure 2.1 Sample Security Policy of Tripwire

```
DIRECTORY=/home/thesis/tripwiretest;
```

```
TWPOL=/etc/tripwire;
TWDB=/var/lib/tripwire;
SIG_MED      =66;


#Policy, three files. One of data, one invariable binary, one growingonly

(

  rulename = "Thesis",
  severity = $(SIG_MED),
  emailto = root@127.0.0.1

)

{

$(DIRECTORY)/binary           ->$(ReadOnly);
$(DIRECTORY)/data             ->$(Dynamic);
$(DIRECTORY)/growing          ->$(Growing);

}
```

Using a database to insure integrity checking may lead to the following problems:

- The verification of the integrity is conducted at certain intervals, usually every day; by computing the digest for a large number of files the integrity checker will stall the computer for some time.
- Between checking periods an attacker could change the file-system, reverting the changes before the integrity checker performs the verification.
- The databases, configuration files and policy file should be strongly protected, possibly in special types of storage.
- If files are changed or new files are added or deleted, the database has to be updated. In large systems, this could be difficult to administrate. Moreover, as the database should be stored in a special storage, updating it may not be straightforward.

- Policy files and configuration files will need to be carefully created and updated. The efficiency of this integrity checking system is strongly linked to the accuracy used when creating the policy file and the competence of the system administrator to handle the reports.

Examples of integrity checkers using the database approach are Tripwire [13], [14] and AIDE [20].

In conclusion, database integrity systems are highly portable and configurable, but they are not real-time integrity detection methods and are difficult to administrate.

2.5.2 File signing

The second category is to use cryptographically signed files to ensure the integrity of the files. This is accomplished by using digital signatures as introduced in the first chapter. When the file is compiled and linked, a digest of the message, such MD5 or SHA, is computed. Then the message digest is encrypted using the creator's private key. The digest is embedded into the file being signed, and the header of the file is changed accordingly. Hence, we will have a signed version of the file.

Another way to perform file signing is to store the file signature in the filesystem itself. File system usually store information about the file in data structures. For example, in the case of the Linux file systems, these data structures are called inodes. Each file has an inode and is identified by an inode number (i-number) in the file system where it resides. inodes provide important information on files such as user and group ownership, access mode (read, write, execute permissions) and type. In novel file sytems these data structures include a pointer to a metadata section, which includes unspecified information about the file. This metadata section could be used to store the digital signature.

The signed file can then be distributed to different machines which supports this feature. Before the file is opened, the kernel first will check that the file is correctly signed, using the creator's private key. If the checking is correct, we can assume that the file has not been corrupted and has been created by the owner. If not, either the file is a forgery, or has been corrupted.

The principle is very simple but the implementation of this integrity method is fairly complex. Several problem arise:

- This method is only to be used on files that are not supposed to change. In fact, this method is mainly designed to ensure the integrity of binaries and possibly other sensitive files such as shared libraries.
- This method is not portable between architectures. Different architectures use different file-system types and compilers. Embedding the signature and patching the kernel to verify the signature will change from operating system to operating system.
- Files usually are not statically linked. Files such as these, called dynamically linked executables, use some functions declared in libraries that are common to the system. Hence, not only the integrity of the file should be checked, but also the integrity of the shared library. This issue is discussed in [12].
- As the file has to be checked before the execution, the execution time of a signed file will be longer than a normal file. This performance penalty can be decreased using caches or buffers, as discussed in [12] and [16].

To conclude, this system detects corrupted or forged files in real-time, but is not portable and cannot be used in general files or configured to meet some requirements. The use of this approach may involve a performance penalty in the execution of files.

Chapter 3

3 Out of band verification system

In the previous chapter Integrity Verification Systems were analyzed, revealing that the use of Integrity Verification System will add another layer of security to the system.

However, all Integrity Verification Systems share a common problem. To complete their tasks, Integrity Verification Systems must trust the operating system to operate as anticipated. This assumption is not always valid, as this file integrity system checkers are designed to detect an intruder that possibly has already gained supervisor privileges. With such privileges he or she may be able to change the behavior of the operating system, hence leading the system to false negatives.

To overcome this problem the inspection of the system's integrity should be performed by completely trusted, non-corruptible software even in the event of a successful intrusion. This goal could be achieved using an out-of-band system as the auditor. In the following sections this system will be described.

3.1 Motivation

A security system that relies on the operating system of a penetrated system cannot be trusted. In fact, this problem is well known. In an article, which appeared in Phrack Magazine, signed by "Halflife" [25], a loadable kernel module was used to bypass the Tripwire integrity checking system. Since then, several tools for corrupting the operating system have been developed including Knark, famous for being used in the Ramen worm

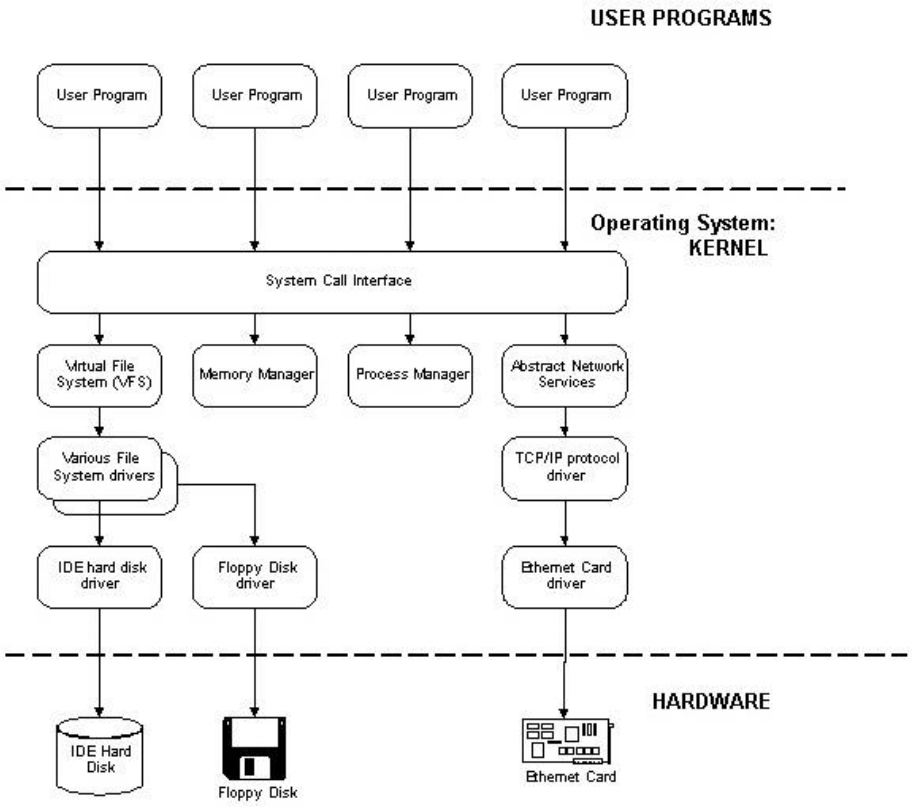
[42]. This section will explain how these attacks works. The attacks explained are under Unix-like operating system. However, other operating systems are also vulnerable to these kinds of attacks as will be explained at the end of this section.

Before discussing the actual attack, some background is needed. In the next two sections the operating system calls and the feature of dynamically changing the kernel using Kernel Modules in Unix-like system will be discussed.

3.1.1 System Calls

User processes and the kernel run in different modes. The CPU itself enforces this policy. Every modern processor has at least two modes of operations, and in some cases, as in the x86, more than two. Every mode of operation has some actions allowed and others not permitted. In the case of the Unix operating system, only two levels are used, the higher, and the lower. The lower lever is called user space or protected mode. In this mode the user has restricted access to the memory and hardware devices. The higher level is called Kernel space or supervisor mode. In this mode the process has unrestricted access to memory and devices. User-space applications are run in protected mode, while the kernel execute in supervisor mode. The only way an application can access the sources restricted by the protected mode is through the Kernel.

Figure 3.1 Linux Kernel Space structure. From [45]



If an application requests a service from the kernel, such as asking for more memory or accessing a hardware device, system calls are used to access the second mode of operation. The only way to access kernel space is by using system calls or when a hardware interrupt arrives to the system. In order to use system call the process will fill certain registers with appropriate values, include the type of system call to access, and call a defined interrupt, dependent on the operating system and architecture. For example, in the Intel architecture the user process will call interrupt 0x80 if the operating system is Linux or interrupt 0x21 if the operating system is Windows. Then, depending on the system call used, the process will jump to a certain location of the kernel. The location in Linux is stored in a table (sys_call_table), where the addresses of the functions in the kernel to be called are stored.

The kernel will look at this table and jump to the corresponding address in the kernel. After it returns from the call the kernel will do some system checks and continue in the address of the user space calling process.

3.1.2 Loadable Kernel Modules

A feature of the Unix-like Operating Systems is the possibility of dynamically changing parts of the kernel. This is performed using LKM (Loadable Kernel Modules). When requested, the module code resides in the kernel's address space and executes entirely within the context of the kernel. Command lines functions can be used to load and unload the modules. In Linux these commands are insmod, for loading, and rmmod, for unloading. Also, they can be loaded automatically by a daemon. A daemon is a program that runs continuously and exists for the purpose of handling periodic service requests that a computer system expects to receive. In the Linux case, the daemon used is named kerneld. As part of the kernel, only users with administrator privileges can load and unload modules. However, kernel modules, as they are part of the kernel, when used with malicious purposes can be a powerful tool. There are several legitimate uses for kernel modules. The most important use is to implement device drivers. Statically linking in the Kernel image all the devices drivers could result in a Kernel which is unnecessarily large if these devices are used only at specific times, such as USB devices or PCMCIA cards. These drivers should be loaded only when the device is to be used. LKM can also be used for testing purposes or for supporting file-systems not normally used.

Loadable Kernel Modules are part of the kernel, and should be programmed and compiled as such. Loadable Kernel Modules cannot use user space libraries. Also, LKM should be programmed with extra care to avoid bugs. A bug in a user space program will probably

lead to the termination of the process, while a bug in a Loadable Kernel Module could lead to an unstable system and possibly to a system crash.

3.1.3 Attacks in kernel space

An Integrity Verification System which trusts the operating system can be deceived using attacks that involves changing the kernel. These attacks will be explored in the Linux Operating System. Similar attacks could be launched in other Unix like Operating Systems and are briefly discussed at the end of this section.

The straightforward way of changing the kernel is to replace the kernel binary itself. The kernel binary is usually placed in the /boot partition, so an attacker could compile his/her own version of the kernel and replace the binary. This is usually infeasible, for two main reasons

In secure systems, the kernel binary is usually stored in a physically non-writable partition, making it impossible to replace the file

To recompile the kernel with exactly the same option is complex. The administrator will probably notice a new Kernel Image, as the behavior of the system will possibly be faulty.

Another possibility is using LKM. An attacker will not have to recompile the complete kernel, just code a LKM, which will be loaded at boot time and will be part of the kernel.

Once the intruder has gained access to the kernel space, several attacks could be launched against the system to avoid being detected. The most obvious attack is to redirect the system calls. Any program in user space such as integrity checkers will use system calls to access kernel space, even for very simple operation like reading a file. By redirecting the system call to a “rogue” system call the attacker can hide the existence of any file in the system even from integrity checkers. Redirecting a system using kernel modules is simple.

As we have seen, the addresses to jump to when a system call is loaded are stored in a table. When the module is initialized, the kernel module will use code similar to the following

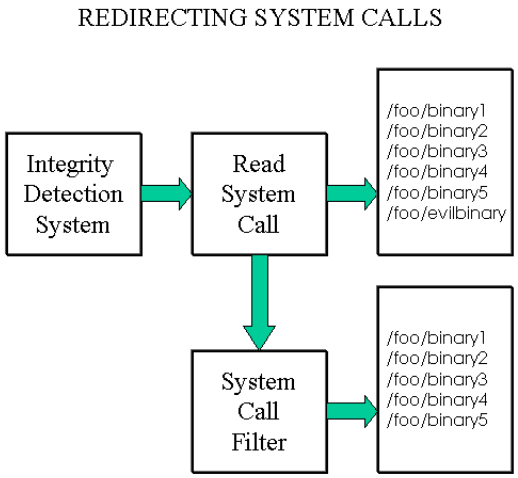
```
original_syscall=sys_call_table[SYS_sycalltohack]
sys_call_table[SYS_syscalltohack]=hacked_syscall
```

Where hacked_syscall is a pointer to the function used to replace the system call.

In the function hacked_syscall the attacker will call the original syscall and the change the results. As an example

```
res=(*original_syscall)(parameters)
//change res to mislead the system
return(res)
```

Figure 3.2 Redirecting system calls



Several Rootkits, a set of tools that an attacker uses to mask an intrusion and/or regain access later, take advantage of kernel modules. These tools are available on the web using these techniques. An example is Knark.

Other operating systems, such as Windows NT, are also target for these attacks. Malicious patches to the system or corrupted drivers can be loaded to corrupt the kernel. These tools are also available on the web.

Some efforts have been made to counter the loading of kernel modules. Most of these techniques, as [43] in Windows NT, operate by avoiding modules or drivers to be loaded. However, an intruder could just reboot the machine. This will delay the call to these programs. The attacker will then load the malicious kernel modules or drivers before these tools begin performing the checks. Another method to bypass this type of security measure is to directly access the physical memory and load the module.

3.2 Concepts, terminology and assumptions

In the present and the following sections the terms host processor or host system will be use to define the machine or set of machines to be audited or verified for correct functioning. The term “host” is slightly inaccurate, as the implementation of the independent auditors could be diverse. However, as in this work, the out-of-band verification system is implemented as an embedded coprocessor, and the term host processor will be used in order to avoid confusions. Other terms, such as host operating system, will be used throughout the text to refer to components of the system to be verified.

System A will be called out-of-band auditor or independent auditor of System B if it accomplishes the following set of properties

i. Unrestricted access: Machine A must have unrestricted access to the internal devices of machine B to be verified or needed for the verification, including peripherals, hard disks and interrupts. Notice, however, that unrestricted write access to the internal components of the host system could lead to an unstable system. Hence, the independent auditor should use the write access to the components of the system only in cases of a high-risk alert to avoid further damage to the system.

ii. Secure transactions: The channel used by the independent auditor to retrieve the data should be a secure channel. A secure channel is a channel, which cannot be eavesdropped or intercepted.

iii. Inaccessibility: Machine B must not have access in any way to the internal components of machine A, including memory and internal interrupts.

iv. Continuity: Machine A must run immediately after machine B has setup the internal devices and is in a known trusted state. After this moment, Machine A must run continuously, independently of the behavior of machine B. Notice that power failures or hardware reboots should be the only way to restart machine A and must be labeled as high risk level alerts.

v. Transparency: The access to the internal devices should be transparent to the host system. However, concurrent access to the devices will probably occur. In these situations, the consequences to the host system should be minimized.

vi. Verifiable software: All the code running in machine A must be trusted and verifiable. This, at least, implies that all running software in machine A must have

the source code available. This includes the firmware, operating system and user space programs in machine A.

vii. Non-volatile Memory: Machine A must be capable of retaining a record of the alerts even in the event of a power failure or reboot. Hence, machine A should have some sort of non-volatile storage to record sensitive data.

viii. Physically secure: Finally, machine A should be physically secure. Mechanisms to accomplish this requirement are discussed in section ensuring the physical security of the system.

Further discussion in this chapter assumes that the independent auditor meets some or all these requirements.

3.3 Modes of operation

The independent auditor could be in three different states. The normal state is the normal mode of operation, and is dependent on the method used for the integrity checking. The second state is the alarm state. This state will be discussed in the section 3.4, and is reached if an alarm is triggered in the normal state. Another mode of operation, which can be accessed only at boot time, is the management mode. This mode is only accessible through a set of secure mechanism, and will allow the administrator to change parameters in the secure coprocessor. This mode is in a secure state, as can only be accesed physically and an independent auditor is physically secure by definition.

The normal mode of operation could follow different methods to ensure the integrity of the data in the host machine. Three methods will be discussed. The first two methods, using a

database and file signing come directly from the approaches of Integrity Verification System. The third approach involves using the independent auditor in a promiscuous mode. All these modes of operations assume that the host operating system is in a trusted state when the first checking takes place. Using the independent auditor in an untrusted state will not add any security to the system.

3.3.1 Objectives

The modes of operation described in the previous section should achieve the following objectives

- Check the integrity of certain files without trusting any software outside the independent auditor
- Whenever an alarm occurs, the independent auditor should be able to send an alert to the administrator
- An attacker with complete knowledge and access to the system, whose only restriction is physical access to the hardware of the independent auditor, will not be able to bypass the integrity system
- The integrity system should be able to work in real-time. If this is not possible, the independent auditor should be able to store sensitive data of the machine before the attack.

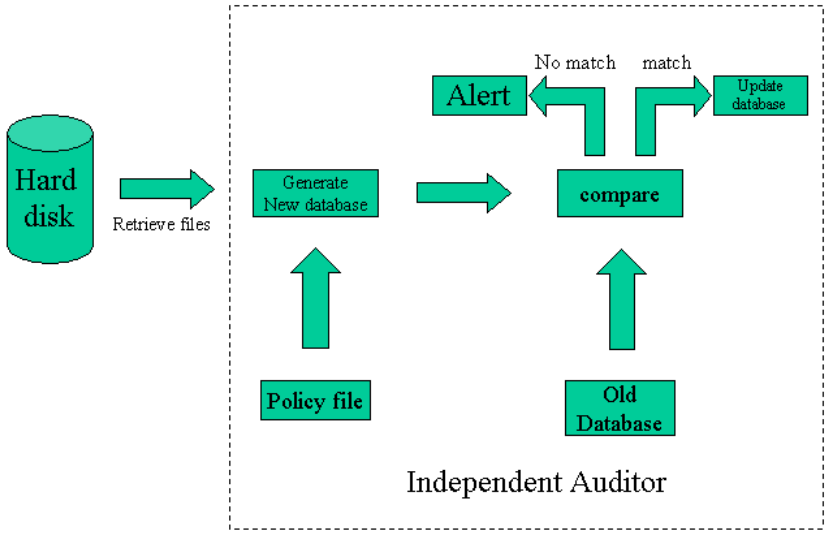
3.3.2 Independent auditor in database mode

The concept of using a database for integrity checkers was described in the previous chapter. In this section this concept will be reviewed in order to be used with an independent auditor.

In this case the independent auditor will have a policy file, which is uploaded into the system in management mode, where the files to be checked will be declared along with the parameters of the checking.

The files will be accessed every certain period of time, and the set of actions stated by the policy file will occur. The independent auditor will retrieve information of the file, possibly computing the hash function. This information will then be checked against the locally stored information inside the auditor. If the information matches, the new information will be stored in the nonvolatile storage. If they do not match, the alarm state is triggered.

Figure 3.3 Database mode of operation



Using an independent auditor in database mode has several advantages as compared to its counterpart managed by the host operating system.

The auditor handles the computational work. Hence, stronger hash functions can be used to ensure the integrity of the database without decreasing the performance of the host system

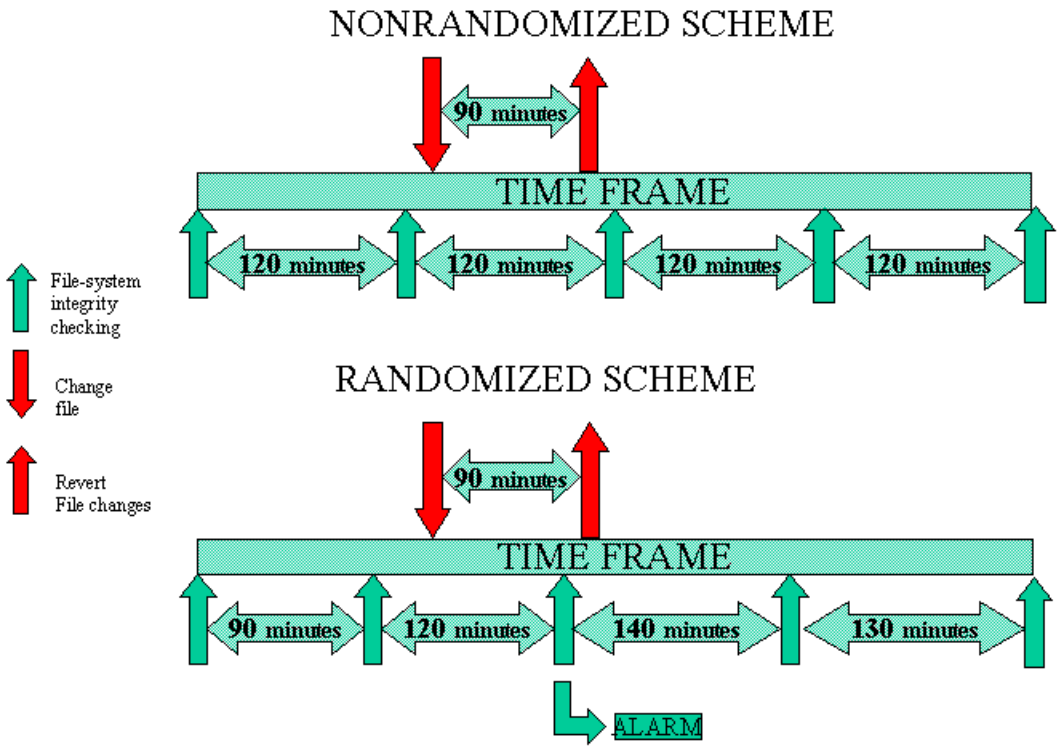
The system is not vulnerable to subversion errors. An attacker gaining access to the host machine will not be able to corrupt the auditor, as by definition, the machine B cannot have access to any data from the auditor

The system should be able to securely log files and other sensitive data in the case of an alert. The methods and necessity for secure audit logs will be discussed in depth in section 3.5

Although all the computational work is done in the auditor, accessing the file-system could still decrease the performance of the host computer. A mechanism is needed to make the access from the auditor to the drive transparent.

As the security checking should still be scheduled at certain times, an all-powerful attacker could corrupt the file-system between checking and revert to the initial state just before the next inspection. Such an attack would remain unnoticed by the independent auditor. To overcome this attack, the schedule time for the integrity checking should be randomized. In management mode the administrator will set parameters such the average and the variance between inspections. Using this mechanism, even and all powerful attacker will be unable to retrieve information about the next checking, even with administrator access to the host machine and knowledge of the last checking time.

Figure 3.4 Randomized scheme vs. nonrandomized scheme



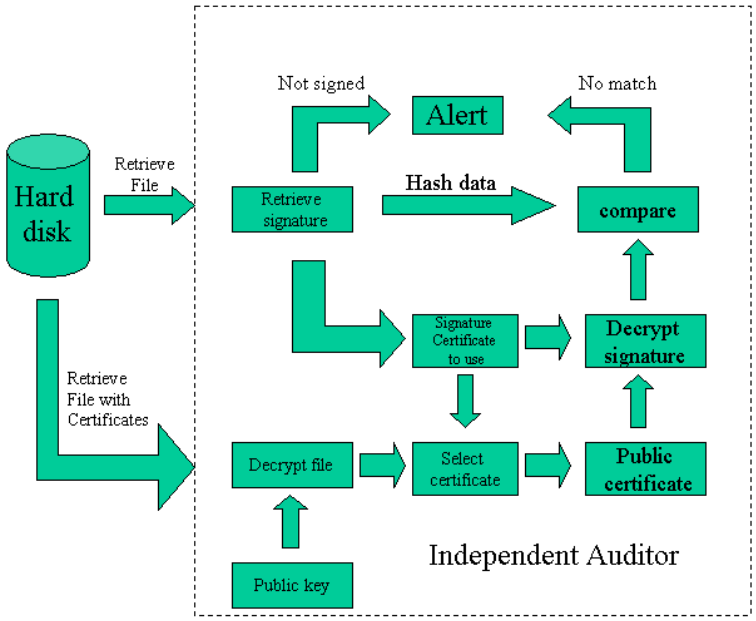
3.3.3 Independent auditor in file signing mode

The file-signing mode using an independent auditor differs from its counterpart using the host operating system. In this case, is not possible to achieve a real-time approach, as the auditor will not be able to tell which files are going to be executed without communicating with the host processor. Such communication is possible using different techniques, for example, the auditor could poll a special file with information about the status of the operating system; These techniques rely on information provided by the host operating system, and should be avoided, or at least not trusted. However, hybrids methods could be used.

In file-signing mode the auditor will retrieve all the files in a given partition at every scheduled time. All the files in the given partition should be signed. Any corrupted or unsigned file will trigger an alarm. As in the case of the database mode, this checking should happen at random times.

If the file signing is implemented using the metadata parameter of the file-sistem data structures, the independent auditor should match the file with the data structure and fetch the information stored in the metadata.

Figure 3.5 Independent auditor in File Signing mode



In the hybrid mode, the auditor will receive information about the host using a shared register (for example, mailbox registers). This could be easily achieved by using a kernel module in the operating system. This information could include new files in the system, files to be executed or shared libraries, to mention some. The auditor will react

consequently, checking the new files or the executed file. As appealing as this method could be, an attacker could just use another kernel module to send erroneous information to the auditor. Hence, the auditor should still verify the partition every scheduled time to ensure that the system has not been compromised. A possible attack could be to send erroneous information to stall the auditor (for example, fake the execution of a file every second) so the auditor will be busy attending these requests. To avoid this attack the scheduled verification should have priority against the information sent by the host operating system

Another important issue is the key management. In this scheme, the auditor will have a public key stored, which can only be changed in management mode. The certificates used to sign the files will be stored in a special file in the partition where the signed files reside. This special file will be signed with the private key, which is in the possession of the administrator only. At boot time, the auditor will retrieve the certificates from this file using the public key. Two points to be noticed are:

- Adding or revoking certificates will change the content of the special file.
- The public key inside the auditor is not a secret. Retrieving the certificates will not give any advantage to the attacker. The only secret is the private key maintained by the administrator.

As in the case of the database system, log files could be stored and retrieved in the case of an alert. This will be discussed in section 3.5

3.3.4 Independent auditor in promiscuous mode

In the previous section it has been stated that the auditor will not know when the host processor is executing a file. However, the auditor could infer actions of the host processor

by examining certain hardware component, for example interrupts or registers, in the peripherals.

As in the previous section, the files to be audited should be signed. The auditor will be observing the behavior of the system, and if it infers that a file is being used, it will inspect it.

For example, the auditor could be “listening” to the IDE interrupt controller. The IDE controller will raise an interrupt when a file is written, and a register will be updated. The auditor will “listen” to this interrupt, or poll the register to know what kind of action (read, write) the host processor is requesting.

A read could be either an execution or a reading. From the point of view of the peripheral both actions are the same, requesting a file from the disk. Also, the request could be partial, as part or the entire file could be in the internal cache of the host system. This makes it difficult for the auditor to react to the knowledge of these actions. A write signal, on the other hand, is easier to detect. The auditor can deduce if it is a new write or change to an existing file. In the first case, the auditor should check the integrity of the file once the transmission has finished. In the second case, the auditor should trigger an alarm, as signed files should not be changed.

3.4 Alerts

In this section the alert mode will be described. Any alarm should change the auditor to alert mode. The reactions to an alarm depend on policy. The policy file regarding the alert should be created in management mode before starting to use the auditor. The policy file should match the type of alarms (reboot, corruption of a file, not signed file, etc) with an

alert. Although policy files could be implemented in different manners, in this section three possible type of alerts to an alarm will be discussed.

The auditor could trigger physical alerts for critical security problems. Physical alarms are the ones that directly affect the hardware of the system or that trigger a device outside the system. This is possible as the auditor has access at least to the device to be audited, and hence can shut it down for example. This is an extreme solution, but for very sensitive systems such as military system, taking the chance of an intruder retrieving or corrupting data from the system could be unacceptable. Physical alerts have the obvious drawback of being the perfect target for Denial of Service attacks. However, if the intruder has arrived to a point where it could trigger this kind of alarms, this eventuality is a minor evil.

Other physical alerts, which are not as severe, could be that of printing a message to the screen or sounding an audible alarm.

The auditor could send the alerts through the network to an external trusted machine or system. These types of alerts are network alerts. A secure channel must exist between the auditor and the external trusted machine to avoid an eavesdropper to disrupt the communication or forge messages. If the channel used is not physically secure, or even is the same channel used by the audited system, a set of secure protocols should be used. That mechanisms may include cryptography to avoid eavesdroppers know sensitive data, and authentication mechanisms to ensure the data send by the auditor is not corrupted. The auditor should send data alerts even if there are no alarms to counter the attack of an intruder disrupting the channel.

3.5 Audit Logs

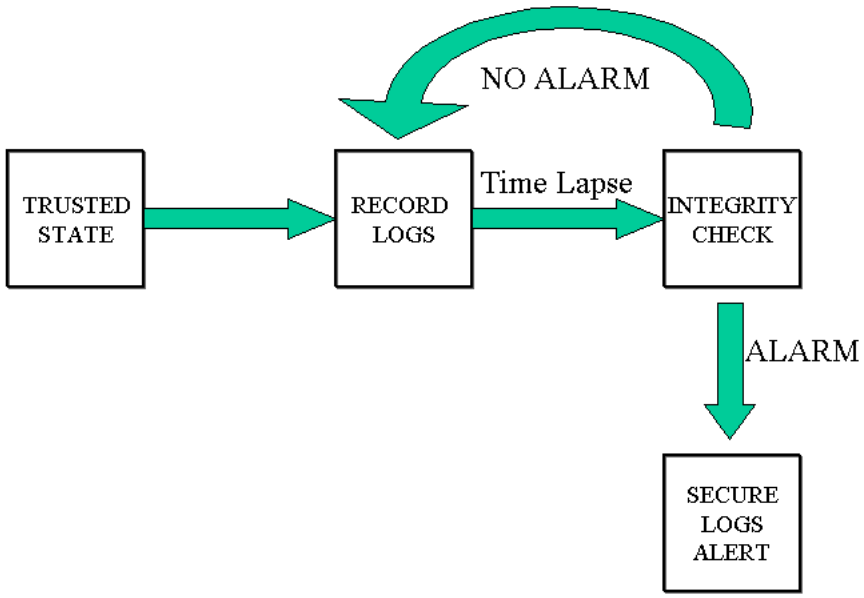
The last set of actions is logging the alarms. Information about the alarm should be stored in the non-volatile storage device in the event of any alert.

Using an independent auditor for integrity also opens the possibility of not only storing the information of the attack but information before the attack. This is useful in the modes where the auditor is not checking the system in real time, as it happens in the database approach. The auditor could log processes, measurements or events. The auditor stores these sensitive logs in a trusted state. The first checking is always assumed to be in a trusted state. Every checking without an alarm will ensure that the system remains in this trusted state. Hence, after the system audit took place, the system will store the sensitive data. In the next audit without an alarm, the auditor will update the data. In the event of an alarm, the data before the system compromise took place will be preserved, allowing the supervisor to retrieve the logs before the attack took place, in a trusted state. The recorded file could be compared to the files to know if the attacker has tampered with the logs, and possibly information about the type of attack and identity of the intruder. A discussion of the importance of secure audit logs can be found in [11].

The following figure represents the logging procedure in the independent auditor.

Figure 3.6 Audit Logs procedure

SECURE LOG DESCRIPTION



Chapter 4

4 Implementation of an independent auditor

In this chapter a possible implementation of an independent auditor using an Intel ebsa-285 will be discussed. We will demonstrate that this implementation pursues the properties declared in chapter 3 for independent auditors.

In the first section the hardware used for the implementation will be introduced. Then we will demonstrate that this hardware is suitable for being an independent auditor. Next, the mode of operation of the independent auditor will be explained. In section 4.3 the software used for implementing the independent auditor with the EBSA-285 will be presented. Finally, the operation of the EBSA-285 as an independent auditor will be discussed.

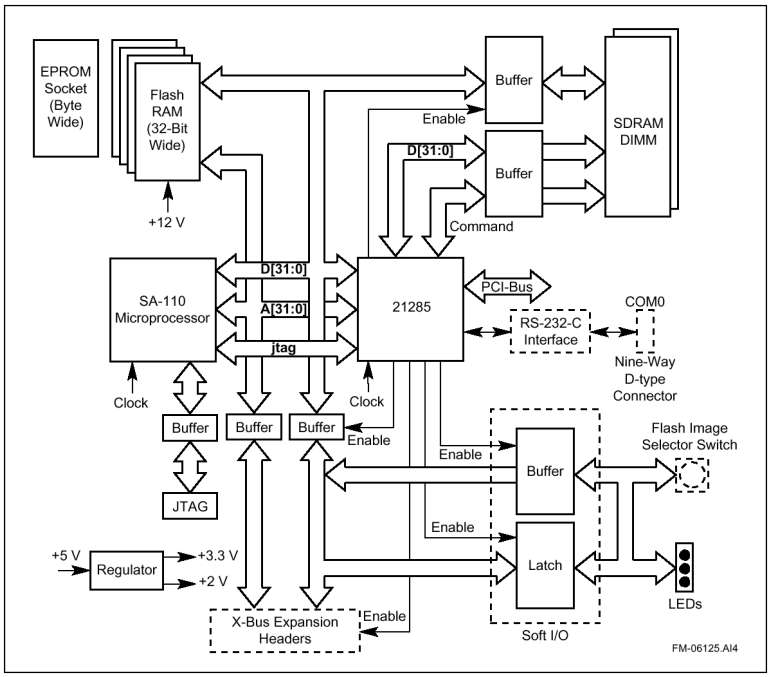
4.1 Hardware description

The information of this section is extracted from the reference manuals [47], [48] and [48]

The hardware used to build the independent auditor is the StrongARM EBSA-285 Evaluation Board. An SA-110 ARM processor, Intel 21285 system core logic, diverse types of memory, different buses and several I/O devices compose the board. In this chapter the board will be referred as the EBSA-285, the processor of the EBSA-285 as the SA-110, and the core logic as the 21285 core logic. The system's main processor will be referred to as the host processor, or just as the host. The devices plugged into the PCI bus will be referred to as PCI cards or as PCI devices.

The board has 3 LEDS that can be used for diverse purposes, a switch with 16 positions used to choose the memory Image in the flash memory (discussed in section 4.x), a 9-way D-type serial port, a JTAG connector, used for debugging purposes and a port for a possible daughter board.

Figure 4.1 Structure of the EBSA-285, from [47]



4.1.1 The SA-110 processor

The SA-110 processor is an implementation of the ARM Version 4 architecture. The SA-110 is a 32-bit general-purpose microprocessor with a 16 Kbytes instruction cache, a 16 Kbytes write-back data cache, a 128 bytes write buffer, and a memory-management unit (MMU). The processor can function either with 32 bit or 26 bit instruction sets. The processor has 16 core clock frequencies ranging from 88.3 MHz to 287 MHz. The power required by the processor is 2 volts.

4.1.2 The 21285 Core Logic for SA-110 Microprocessor

The Intel Semiconductor's 21285 is a single-chip interface between the SA-110 processor, an SDRAM memory, flash memory and the PCI bus. Some interesting features of this interface are programmable timers, Doorbell register, which could be used by the host processor or any PCI device to communicate with the EBSA-285, I2O message unit, also used by the EBSA-285 to communicate with the host processor (as will be seen in section) an X-bus, power management support, DMA controllers and DAC support. It supports both 5-V and 3.3 V signaling environment.

4.1.3 Operation modes of the EBSA-285 board

The EBSA-285 is a single-board computer in the form of a PCI plug-in card.

This board is designed to function in one of two different modes. The first mode is the host bridge mode. A PCI based architecture has a multi-master capability, allowing in PCI master to access any other PCI master/target. One device in the PCI is responsible to for generating a software-driven initialization and to configure all devices after power up or reset. This PCI device is named the Host Bridge. In this mode of operation the EBSA-285 acts as the host bridge of the system and the SA-110 as the main processor. As host bridge the EBSA-285 should provide PCI configuration cycle for the devices plugged in the PCI bus, bus arbitration, PCI clock generation, interrupt controller, reset generation and pull-ups on some bus signals. To be used in this mode, the EBSA-285 should be plugged into a special slot of the motherboard (the "system" or "Host" slot) and some jumpers should be

configured. This mode of operation is not suitable for building an independent auditor, and hence will not be used.

The second mode of operation, called add-in card mode is the mode to be used in the implementation of an independent auditor. In this mode of operation the motherboard or PCI backplane provides bus arbitration, interrupt controller and pull-ups for the system. The SA-110 is a coprocessor of the system in this configuration. The Host CPU is in charge of configuring the PCI devices. The EBSA-285 appears to the host system as a PCI device. The SA-110 is able to access the system memory as PCI bus master. The SA-110 may allow the host CPU to have a window of the memory in the EBSA-285 mapped in the PCI memory.

There is another mode of operation, called blank programming mode, which maps the local Flash memory into the PCI bus to be rewrote. This mode of operation will be discussed in the next section.

4.1.4 Memory specifications

The EBSA-285 includes support for both volatile and non-volatile storage. The non-volatile storage is supplied in the form of flash ROM memory, which has been explained in the first chapter. This flash memory is divided in four 1 MB flash ROMS. These 8 bytes wide flash memories are arranged to provide a 32-bit path for the SA-110. Each flash ROM is subdivided into 64 KB blocks of memory. Because of the arrangement of the memory to provide a 32-byte path, the whole flash memory can be treated as sixteen 256 KB blocks. If the firmware that is shipped with the EBSA-285 is used, any of these blocks can be selected to be executed using the 16-position switch. The independent auditor will use its own firmware, so this switch will not be used. The blocks are contiguous, so they can be

either flashed as a whole or separately. To program the flash ROM the SA-110 must not be executing from the Flash ROM, because the flash programming requires a well-defined series of read and writes to the flash and code fetches will disrupt this sequence. The 21285 allows the flash to be reprogrammed from the PCI interface while the SA-110 is running. However the EBSA-285 has a special mechanism to reprogram the flash called “Blank programming mode” or “Blank ROM mode”. In this mode the EBSA-285 allocates the resources and maps the ROM in the PCI bus, which allows the host processor to write on it. Also the EBSA-285 can be configured to accommodate an 8-bit ROM emulator.

As a volatile storage the EBSA-285 uses SDRAM memory. The SDRAM memory can be attached to two 168-pin 3.3.V SDRAM DIMMSs. The EBSA-285 is shipped with a 16 MB DIMM.

4.1.5 Buses

There are seven main buses in the EBSA-285, which includes: CPU Address bus, CPU Data Bus, Buffered Data Bus, X-bus, Buffered X-bus Data Bus, SDRAM address bus and buffered SDRAM address bus. Both he CPU address bus and the data bus connect to the SA-110 processor, the 21285 system controller and the flash ROM. The SA-110 uses this bus to drive addresses from SDRAM, flash ROM and 21285. The X-bus is a sub-set of the CPU address bus, which is always enabled. It is a low speed bus used to access I/O devices. The SDRAM address bus is permanently enable and is used to drive the information from and to the SDRAM DIMM sockets.

4.1.6 PCI interface

The PCI bus in the EBSA-285 is a full PCI local Bus Specification, Revision 2.1, 32 bit, 33 MHz compliant interface. The bus is provided to allow the EBSA-285 to be used as an add-in card. Signaling levels used are 3.3 Volts although the interface is 5 Volts tolerant. In central function the PCI interface is capable of using a number of reserved pins to provide four sets of interrupts, requests and grant pins required. In add-in mode, however, these pins are reserved and the PCI card is only able to use a pin to raise an interrupt with which it may communicate with the host processor. As PCI bus master, the SA-110 is capable to read the entire PCI memory through the PCI interface.

4.2 The EBSA-285 as an independent auditor

In this chapter the suitability of the EBSA-285 to act as an independent auditor will be discussed. In section 3.2 we stated the properties to be accomplished by an independent auditor. As an assumption the Host machine should be physically secure. The term physically secure is used to describe a machine which internals part could not be tampered with by an attacker. The attacker, however, could have access to the peripherals attached to the host, including keyboard and monitor. Hence, the physical security property will be declared as an assumption.

- The unrestricted access property is accomplished using the PCI interface. In our case, the EBSA-285 must have unrestricted access to the hard disk data and to the Ethernet controller. The host processor configures at boot time all the PCI devices. The EBSA-285 has access to the entire PCI Bus, hence is capable to read the register and data from all PCI plugged in the same bus. Notice that if the EBSA-285 is plugged in a slot using a different bus than the ebsa-285, the access will not be

possible and the EBSA-285 will not be capable to act as an independent auditor of the host processor. The EBSA-285 is not able to “listen” to the interrupts raised by the different devices in the PCI-bus. These interrupts, however are not imperative to the auditing, as a polled method could be used to read and write data to the peripherals.

- The channel used to retrieve the information from the peripherals is the PCI bus. This channel is secure as it is an internal part of the computer and is supposed to be secure. Hence, the EBSA-285 accomplishes the secure transactions property.
- The inaccessibility property is met if the bootloader (see section 4.3.2) acts in standalone mode. In this mode the EBSA-285 does not map its meory (either ROM or RAM) in the PCI bus, allowing the host processor to access only mailbox registers and PCI registers. Accessing this register does not influence the EBSA-285, hence not breaking the inaccessibility property.
- Once the EBSA-285 starts auditing only a power failure or reset of the host machine will stop it from functioning, and will be labeled as alarms. The EBSA-285 will begin functioning after the host machine have set up all the internal peripherals. In our case, the EBSA-285 will begin functioning before this happens, so the EBSA-285 should have a mechanism to stall its booting until all the peripherals have been configured. This will be discussed in section 4.3.2. Therefore the continuity property is proven.
- The EBSA-285 can access directly the register of the PCI devices, so accessing the data could be performed by the EBSA-285 without supervision of the host operating system using polling, hence satisfying the transparency property. Notice,

however, that some mechanism should be implemented to avoid concurrent writes to the register, which would lead to an unstable system. This will be discussed in the next chapter.

- The software running in the EBSA-285 is open source. It is composed by a minimum bootloader, the Linux operating system and open source software. The code is hence verifiable. To be trusted, the code should come from a trusted party. To ensure that the code is from the claimed party, architecture similar to the SEBOS architecture could be used [46]. However, the implications of trust go beyond the purpose of this work. In our case, we suppose the Linux operating system as trusted and verified. The remaining code has been verified, therefore is also trusted.
- In chapter 4.1.5 the different types of memory in the EBSA-285 were stated. The EBSA-285 can use the flash ROM to store the alarms and logs. The ROM cannot be reprogrammed if the program executed from the flash ROM. To avoid this problem, the bootloader copies the root file-system and the operating system to the RAM memory before executing it, freeing the flash ROM.

4.3 Software in the ebsa-285

The EBSA-285 is composed of a bootloader, a port for the ARM architecture of the Linux operating system and user level programs stored in a Ramdisk (a file system resident in the RAM memory). For building the EBSA-285 some drivers created for the host were used, in order to create interaction between the host processor and the EBSA-285. In the following

sections these different component will be discussed. All the software in this section can be downloaded from [50].

4.3.1 The toolchain

As we have seen in the first chapter, the term host system was used to define the place were the applications for the embedded systems or target were created. In order to avoid confusion with the terminology, the machine where the applications are cross-compiled for the co-auditor will be called the builder system. The embedded system will be still called the target system or just the embedded system. A toolchain is a set of application that allows a builder system that does not share the same architecture with the embedded system to cross-compile the programs to the architecture of the embedded system.

In our case, our builder system is an Intel Pentium 3, and our Target system is the EBSA-285. So a toolchain to crosscompile from the x86 architecture (the name of the architecture of the Intel Pentium 3 processor) to the ARM architecture (architecture of the SA-110 processor) is needed.

The toolchain consists of a number of components. The main one is the compiler itself, gcc, which can be native to the host or a cross-compiler. This is supported by binutils, a set of tools for manipulating binaries. Also, the Kernel header will be needed to cross-compile the Linux Kernel. To cross-compile user space programs the C library glibc will be needed. Building the toolchain is a difficult exercise, as all the tools should match in version, and compiling a compiler is definitely a complex process. At the present time, however, several pre-built toolchains are available through the Web. These toolchains are usually suitable for most of the cases. Regretfully, the Linux Kernel version used for the project (2.2.9) is too old to be cross-compiled using these pre-built toolchains. The components used to build

our toolchain are binutils version 2..9.1.0.25 and egcs version 1.1.2. The version of the C library glibc is 2.1.2.

A step-by-step guide on how to build the toolchain can be found at [50].

4.3.2 The bootloader

The bootloader is the first code executed by a machine after a reset or power off. This code usually set up basic registers and configures the memory and peripherals of the system. The bootloader in the EBSA-285 is stored in the flash ROM memory. Mark Van Doesburg created the first version of the bootlader for the pcimsg utilities, which will be described in chapter [x]. From this bootloader, we created two different bootloaders. The first bootloader is used for debugging and testing purposes, and is used in combination with the device driver for the host described in the next section. The second bootloader is used by the EBSA-285 in its normal mode of operation. The bootloader is divided into assembler code and C code. Both versions can be found in Apendix A. Both bootladers share the first part of code, that follows this steps:

- Store the Vectors where the exceptions will go into the ROM memory. This is done because after reset the 21285 decode the flash ROM in two locations; its normal base address 0x41000000 and 0. After reset, the machine will begin executing at memory address 0. After the first writing operation, this alias is disabled, so the vectors should be transferred to the ROM. For the exacts steps, refer to Section 4.1 in [49].
- Configure the X-bus and SA-110 control status registers. For the exacts steps, refer to Section 4.4 [49].
- Turn on the LEDS.

- Configure the memory. To configure the memory several steps should be taken. The main steps are to configure the 21285 memory registers and to configure the mode registers in the SDRAM arrays. For exact steps refer to section 4.7 of [49].
- Copy the first part of the bootloader to RAM and jump to it
- The next part is written in C. In this part both bootloaders differ. The first bootloader, used for testing purpose, will map the SDRAM memory of the 21285 in PCI memory. Then we would be able to pass the Kernel and Ramdisk using the host processor. This feature is very interesting, as flashing the ROM is time consuming, and involves changing jumpers in the EBSA-285 to set the blank programming mode and using the software to flash the memory. Of course, this bootlader is not suitable to be used with the EBSA-285 as an independent auditor, as it will allow the host processor to access to the EBSA-285 memory, conflicting with the inaccessibility property. The bootloder will follow these steps
- Map all the RAM in PCI space.
- Disable the access from the PCI to the ROM.
- Request an interrupt line to the PCI.
- Set the INITIALIZATION_COMPLETE bit in the SA-110 control register. This allows the host processor to read the information of the host such as Vendor and device ID. This bit should be set or initialized, as the host processor will attempt to access the EBSA-285 PCI configuration registers as part of the power up self-test (POST). If this bit is not set, the host processor will retry. If the bit is never set, the host processor will retry forever, thus hanging the system.
- Light the green led. This is useful to know if the bootloader has reached this point.

- Test a mailbox register to continue. The mailbox is initialized to zero. The bootloader will stall here until this mailbox is set to 1. The host processor using the PCI bus can always access the mailbox registers. The host processor is in charge of changing this mailbox register to continue execution. Notice that accessing the mailbox registers does not break the inaccessibility property, as changing this registers does not interfere with the operation of the EBSA-285.
- Turn off the green LED to ensure the mailbox register was correctly updated by the host.
- Jump to the location where the operating system is stored.

This bootloader is complemented with the drivers for the host described in the next section. This bootloader will be called the wait-for-host bootloader.

The bootloader used for the EBSA-285 as an independent auditor continues as follows

- Disable access to the SDRAM from the PCI bus to ensure the inaccessibility property.
- Disable access to the ROM from the PCI bus, to ensure the inaccessibility property.
- Light the green LED.
- Set the INITIALIZATION_COMPLETE bit in the SA-110 control register to 1.
- Copy the Kernel Image from a configured address in ROM to a configured address in RAM.
- Copy the Ramdisk from a configured address in ROM to a configured address in RAM.
- Copy a configuration file from an address in ROM to an address in RAM. This configuration file is used to pass parameters to the Kernel. The configuration file is

not strictly necessary, as these parameters could be hard coded in the Kernel Image. However, as these parameters change frequently, it is useful for debugging purposes to change these parameters without recompiling the Kernel.

- Switch off the green LED.
- Jump to the position of RAM where the Kernel is stored. Continue execution there.

This bootloader will be called the standalone bootloader.

The bootloaders are simple. Almost all the hard work of configuring the PCI, set up the serial BUS, etc is performed by the Linux Kernel.

4.3.3 Device drivers

In the Linux operating system all the peripherals are treated as a normal files. Every device is matched with a special file in the /dev/ directory. These files are associated with a MAJOR number and a MINOR number as well as with the type of device, which could be either a character device, block device or network device. The MAJOR and MINOR numbers are registered in the kernel and are used to access the device driver in kernel associated with the device. Usually the MAJOR number relates to the driver to be chosen, and the device driver uses the MINOR to differentiate between two devices of the same time such as two IDE hard disks usually.

The device driver is kernel code, which could be either implemented using Kernel modules, discussed in the previous chapter, or embedding the code in the Kernel. Device driver make the access to a peripheral transparent. The user only writes or reads using the special file in /dev/, and the device driver will write or read to the specific peripheral. Some peripherals, however, uses other mechanisms that cannot be abstracted using read and write calls. For example, changing the baud rate in a modem will not be possible using reads and writes. A

user space program could access these mechanisms using the ioctl system call. The ioctl system call will call the special file, and ask for a special request. As an example, in the modem case would be to change the baud rate from 9600 to 15600. This request will be handled by the device driver, which will access the peripheral, perform the request and return the result to the calling program.

The classes of peripherals handled by the device drivers could be divided in three classes, namely character devices, block devices and network interfaces. A character device is one that could be acceded using a stream of bytes. Examples of character devices are the console or the serial port. The only difference between this devices and a normal file is that moving back and forth in the device could not be allowed.

The second class is block devices. Usually, these types of devices correspond to storage devices such as hard disks, which can host a file-system. The reads and write calls in this case are composed by chunks of data instead of bytes, making the transfer more complex.

Network interfaces are in charge of sending and receiving data packets and are driven by the network subsystem of the kernel. In this case, the device driver is not matched with a file in the file system as happens with both the block and character device drivers. The way to access these devices is to assign a unique name to them. For example, eth0 is used for the Ethernet card. The way of communicate with this class of peripherals differ greatly with the methods used in the char and block drivers. In this case special functions to receive and send packets are called.

Two device drivers were used in the development of this projects.

4.3.3.1 Arm new

The first device driver, written as a kernel module, is named arm_new and is a character device driver. This device driver is intended to be used only with the wait-for-host

bootloader, and it provides the possibility to send the configuration file, the Kernel Image and the ramdisk to the EBSA-285 from the host processor. This driver is a development of a driver created by Mark Van Doesburg for the pcimsg utilities. This driver basically sends the Kernel, Ramdisk and configuration file to the PCI mapped memory of the EBSA-285. Three files in the /dev/ structure of the host file-system are created. These files are /dev/arm0, /dev/arm0_config and /dev/arm0_initrd. These three files share the same MAJOR number, in this case 42, and have different MINOR numbers. When a user level process writes to any of these files, the device driver is called, as arm_new is registered in the kernel with the MAJOR number 42. Then, it uses the MINOR number to know if the user is writing to /dev/arm0, /dev/arm0_config or /dev/arm0_initrd. If the user is writing to the second one, the module will write the data sent to the PCI position of memory schedule to handle the Kernel configuration file for the EBSA_285. If the third file is wrote, then the device driver will write the PCI mapped SDRAM EBSA-285 memory scheduled to handle the Ramdisk. Finally, if the first file is wrote, then data will be sent to the position of memory in the EBSA-285 scheduled to handle the Kernel Image, and the mailbox register used by the bootloader will be updated to one, continuing the booting process in the EBSA-285.

As an example, if we have our configuration file stored in /home/arm/ebsa285/config.file,

the Ramdisk stored in

/home/arm/ebsa285/ramdisk

and the Kernel Image stored in

/home/arm/ebsa285/Image

the following set of commands:

```
cp /home/arm/ebsa285/config.file /dev/arm0_config
```

```
cp /home/arm/ebsa285/ramdisk /dev/arm0_initrd
cp /home/arm/ebsa285/Imege /dev/arm0
```

Will bring the EBSA-285 to initialization if the wait-for-host bootloader is used. Again, notice that this bootloader and utilities are only used for debugging purposed and do not follow the properties for independent auditors.

4.3.3.2 Pcimsg utilities

The second device drivers to be discussed are the pcimsg utilities designed by Mark Van Doesburg.

The pcimsg utilities are network drivers that should be installed in both the host computer and the EBSA-285. These drivers allow communicating with the EBSA-285 and the host computer in a networked fashion. The driver follows the directions 6.3 of [47] and uses the I2O (Input to output) message unit of the 21285. This message unit provides standardized message passing mechanisms between a host and the SA-110 using the PCI bus. The message unit is composed of four logical FIFOs (Firs In First Out queues), two inbound FIFOs and two Outbound FIFOs. The inbound FIFOs manage the messages that are I/O requests from the host processor to the SA-110, while the outbound FIFOs are used to manage messages from the SA-110 to the host processor. The FIFOs are stored in SDRAM, so the I2O capabilities is only usable if at least part of the SDRAM memory is mapped in RAM. Four pointers administered in hardware maintain the FIFOs.

The pcimsg encapsulates the message frame as if it were network packets in the link layer, using IP addresses. The driver will assign an IP address to the host processor and to the SA-110, The sender will form the messages while the receiver will deencapsulate the messages and send the data to the network layer. Using these utilities it is possible, for example, to open a Network File Partition in the host processor readable by the EBSA-285.

4.3.4 Arm Kernel and ARM kernel patches

The EBSA-285 will run the ARM port of the Linux operating system. This port is maintained By Russel King [25]. To be able to use the port first the regular Linux tree, downloadable from [24], should be patched. The process of patching is adding or removing features of code using the difference created by the diff Unix command between two kernel trees. The patch used is available in the arm-linux-webpage [25]. In the case of the ebsa-285, the Kernel used is the version 2.2.9, so the patch used is the one referring to this version However this patch does not support the ebsa-285 as a coprocessor. To add support to the ebsa-285 as a coprocessor, two more patches should be added to the Linux tree. The first one is created by Mark Van Doesburg, and includes the pcimsg utilities. The second patch is created specifically for this project, and adds the possibility for the EBSA-285 to acess the PCI bus memory, which is disabled by default in the regular kernel tree for the EBSA-285 as an add-in card. These patches can be downloaded from [50]. This patch also adds the possibility of add the parameters of the Ramdisk using the configuration file. Once the Kernel is patched, the Kernel should be cross-compiled using the toolchain.

4.4 Operation of the EBSA-285 as an independent auditor

The EBSA-285 can work in two main modes. The first mode is used for debugging purposes, and is called the wait-for-host mode. This mode is not suitable for use the EBSA-285 as an independent auditor. In this mode, using the wait-for-host bootloader, the EBSA-285 will wait until the components (Kernel, configuration file and Ramdisk) are uploaded

by the host using the arm_new module. Once at least the Kernel is uploaded, the EBSA-285 will boot the Kernel.

In the second mode of operation the EBSA-285 will act as an indepenedent auditor of the host processor. In our case, the independent auditor will audit a file-system stored in an IDE-hard disk. To do so, the bootloader will copy the Kernel Image, config file and Ramdisk from ROM, to RAM memory and boot from RAM. The three components must be first flashed in to the ROM memory using the tools described in section 4.1.4

From here, the Linux Kernel gives support for the IDE device driver, and as the EBSA-285 has access to the PCI memory, the data could be retrieved. However, as the IDE driver uses interrupts to make the transfer of data, the IDE device driver should be used in polled mode. Also, if both the host processor and the EBSA-285 are accessing at the same time to the IDE controller registers the file-system could be left in an instable state. The issues about the IDE driver and performance of the system will be discussed in the next chapter.

In this case the EBSA-285 audits the host processor using file signatures. The EBSA-285 will mount the partition where the signed files are stored and check these signatures at every random time scheduled.

Using the pcimsg utilities the host could send information about the transactions in the host processor. As was discussed in the previous chapter, these methods should not be trusted, and only be used as a backup method. The host processor will send only information about the filesystem. The EBSA-285 will never send information about itself or alarms using this messaging system.

Alarms could be sent using an Ethernet controlled if it is plugged in the same PCI bus as the EBSA-285. Alarms could also be sent using the serial port of the EBSA-285. Any

alarms or logs are stored in the ROM memory. A part of the ROM memory is reserved for that purpose. The ROM memory could be written as no programs are being executed from ROM.

5 Checking the integrity of a IDE hard disk using the EBSA-285 as an independent auditor

To accomplish the goal of insuring the integrity of the host processor the independent auditor should be able to read the its filesystem. In this filesystem, the files should be signed as discussed in chapter 3. In our case of study the filesystem is stored in an IDE hard disk. The host system is a Linux bpx, Redhat 7.1, Linux Kernel 2.2.10 and 2.4.2, using an Intel processor Pentium 3. The IDE hard disk is a FUJITSU MPE3136AH, holding a total of 10 Gbytes. The size of the partition where the files to be checked are stored is 23 Mb.

5.1 The IDE controller

An IDE (Intelligent Device Electronics) controller is an embedded controller in the hard disk, which perform the task of controlling the requests from the CPU. The IDE interface integrates both the controller and the drive itself in a single unit. The IDE interface can serve a maximum of two drives, one being the master and one being the slave. High performance IDE drives enable data transfer rates between drive and maim memory of up to 5 Mb/s. On average, however, transfer rates are usually of the order of 3 Mb/s. Higher transfer rates could be achieved other access methods, as DMA

5.1.1 The IDE registers

The CPU is able to access the IDE interface by means of diverse data and control registers. This set of registers is usually called the AT task file. The AT task file is accessed by the CPU using ports ranging from 1f0h to 3f0h. The following table shows the AT task file

Figure 5.1: The AT task file

Register	**Address[bit]**	**Width Write**	**Read/Write**
data register	1f0h	16	R/W
Error register	1f1h	8	R
Precompensation	1f1h	8	W
Error count	1f2h	8	R/W
Sector number	1f3h	8	R/W
Cylinder LSB	1f4h	8	R/W
Cylinder MSB	1f5h	8	R/W
Drive/head	1f6h	8	R/W
Status register	1f7h	8	R
Command register	1f7h	8	W
Alternate status register	3f6h	8	R
Digital output register	3f6h	8	W
Drive address	3f7h	8	R

The data register is used to transfer data between the main memory and the IDE controller. The data in this register is only valid if the the DRQ bit in the status register is set.

The error register is a read-only register, which contains information about the last command issued to the IDE controller.

The precompensation register is only implemented for backward compatibility with an old implementation.

The sector count register can be read and written by the CPU to define a sector to be read, written or verified.

The sector number register specifies the start sector for carrying out a command with disk access. The register contents are updated according to the last executed command, pointing always to the last proceed sector.

The MSB and LSB cylinder registers contains the MSB (Most significant byte) and LSB (Least significant byte) cylinder to be accessed. Two registers are used because the cylinder number is 10 bits long, thus accessing a maximum of 1023 cylinder. However some IDE driver are capable to use the full 2 bytes, thus allowing the access of 65535 cylinders.

The drive/head register is used to determine the drive and head to be accessed in the request.

The status register contains information about the active command. The CPU cannot write to this register. The controller updates the register every command, or if an error occurs. In this register exists a BSY (BUSY) bit indicating that the driver is currently executing a command. If BSY is set, no registers can be accessed except the digital output register.

The command register is used to pass commands from the CPU to the IDE controller. The execution of the command starts immediately after the command byte has been written into the register. Therefore, all the data involving the command should be passed before writing to this register.

Using the digital output register the IDE controller can be set to reset all the drives. Also, these interrupts are used to set the interrupts in the IDE controller. Using this register the

interrupts issued by the IDE controlled can be masked. In this state the CPU may only supervise the controller by polling.

5.1.2 IDE Command phases

The execution of the commands in an IDE interface are carried out in three phases:

- Command phase: The CPU prepares the parameter registers and passes the command code to start the execution.
- Data phase: For commands involving disk access, the drive position the heads and transfers the data between hard disk and CPU
- Result phase: The controller provides information regarding the executed command and issues a hardware interrupt via IRQ14

The CPU writes and reads data using the 16 bits register, and not using DMA (Direct Memory Access).

In data exchange (read and write commands) the controller uses IRQ 14 to synchronize CPU and controller.

- Read sector: The controller enables IRQ 14 when the CPU is able to read a sector. In this case the IDE controller does not issue an interrupt at the beginning of the result phase, thus the number of hardware interrupts is the same as the number of read sectors.

- Write sector: The sectors are transferred from CPU to the hard disk immediately after writing in the command register. The controller activates IRQ14 only at the beginning of the result phase, hence the number of interrupts coincides with the number of written sectors.

5.1.3 The Linux IDE device driver

In Linux the IDE device driver is a block device driver. A general IDE driver, stored in linux/drivers/block/ide.c, and a specific IDE hard disk driver, stored in /linux/drivers/block/ide_disk.c compose it.

When a process requests a file stored in the hard disk, the operating system will search for it first in the internal page cache. The page cache stores the most recently accessed data. If the data is not stored in the page cache, then the operating system will send a request to the IDE controller. In the IDE hard disk, the data could be stored in the cache or in the hard disk. The data in the Hard disk cache or in the disk is transparent for the operating system device driver, but the request will be served much faster if the data is in the hard disk cache.

The EBSA-285 is not able to “listen” to the interrupts sent by the IDE controller. So we should examine a read performed by the IDE driver, an change it to behave in a polled mode.

The first step in a normal read request to the hard disk is to set up the registers. The digital output register is set to normal operation using interrupts. Then the IDE driver will fill the register with the sector, cylinder and head or side of the data to be accessed. Then, it will write the type of command in the command register to be performed by the IDE controller. After this, the device driver will set an interrupt handler. The operating system will invoke this interrupt handler if the interrupt of the hard disk raises an interrupt. However, and expiration handler will get invoked if a certain time expires. In the IDE driver, this time is controlled by the constant declared in the IDE driver header (ide.h) WAIT_CMD. This constant is set to 10 seconds by default.

As we have seen, the EBSA-285 is not able to listen to the IRQ 14 raised by the IDE controller when the data is ready to be retrieved. However, we can use expiration handler to retrieve the data.

This methods works, however it does not give any kind of insurance for transparency to the host processor. If both the host processor and the EBSA-285 try to access the hard disk, a collision will occur, and both will retry the transmission. As the host processor is faster, it will regain control of the IDE controller, and the EBSA-285 will stall until the host processor finish the transaction. However, if we set the WAIT_CMD too low, both the EBSA-285 and the host could interleave operation, leading to a deadlock, in which both machines are waiting for the IDE-controlled to change the status of the control register.

The following table shows the performance of the Host processor using different values of WAIT_CMD. The value HZ is the number of ticks in one second. The benchmark used is hdparm. In the test, the EBSA-285 was running hdparm with direct access to the disk. This leads to the worst-case scenario; both the EBSA-285 and the host processor are reading large amounts of data from the hard disk.

Figure 5.2: Access impact on the HOST of concurrent access

HOST hard disk speed in concurrent access (in mbs second)					
	WAIT_CMD in the EBSA-285				
	No access	HZ	0.8*HZ	0.7*HZ	0.6*HZ
Disk access min (Mb/sec)	3.31	2.81	2.59	2.49	2.34

Disk average (Mb/sec)	3.32	3.11	2.90	2.72	2.55
Cache access (Mb/sec)	68.45	68.45	68.45	68.45	68.45

HOST hard disk speed in concurrent access (in mbs second)					
	WAIT_CMD in the EBSA-285				
	0.5*HZ	0.4*HZ	0.3*HZ	0.2*HZ	0.1*HZ
Disk access min (Mb/sec)	1.96	2.29	1.80	1.89	1.39
Disk average (Mb/sec)	2.35	2.27	2.12	1.90	1.70
Cache access (Mb/sec)	68.45	68.45	68.45	68.45	68.45

Lower values of WAIT_CMD could lead to a deadlock.

An EBSA-285 will take 38 minutes to compute the MD5 hash function of a set of 10 MB files, with a WAIT_CMD of 0*2 HZ. This value is very high. However, this access method is far from optimal. The IDE driver should be changed to avoid collisions to ensure transparency.

Another approach will be use the EBSA-285 using a polled driver for reads. This polled driver achieves a read speed of 2.4 MB/s. The EBSA-285 will take 4 seconds to compute the MD5 hash functions of a set of 10 MB files. However, if the host processor accessed the IDE driver, probably both machines will interleave while writing in the IDE registers. Hence, some locking procedure is necessary.

These results demonstrate the capability of the EBSA-285 to perform IDE-reads even in a worst-case scenario and without changes in the IDE driver.

Chapter 6

6 Conclusions and future work

The use of embedded auditors comes as a necessity after late successful attacks in kernel space. Software running in the machine cannot be trusted if such an attack has been performed, including security software. Using an embedded auditor will defeat these kinds of attacks. No other successful approach has been release to date, and probably never will be.

Using embedded auditors opens a wide range of possibilities, creating a true out-of-band security. In this work we have restricted the study to Integrity Verification Systems. However, building intelligent embedded firewalls or other type of intrusion detection system will increase the security of the machine, releasing the host processor of the time consuming task of ensuring the security of the box. Moving the security tasks to an embedded processor will increase the overall performance of the machine as well as the security, ensuring the security task to be always in a trusted state.

In this work the EBSA-285 as an independent auditor was presented, demonstrating that it can be used as an out-of-band Integrity Verification System. More development should be done with this specific machine, including changing the IDE drivers to increase performance and test the access to other storage devices, such as SCSI hard disks and removable USB devices. The EBSA-285 functions independently of the Host Operating

System. The tests in this work, however, were restricted to the Linux Operating System. Further tests should include other Operating systems such as Windows NT.

Bibliography

[1] C. Kaufman, R. Perlman, M. Speciner. *Network security*. Englewood Cliffs New Jersey, Prentice Hall 1995

[2] M. Barr, *Programming Embedded Systems in C and C++*. Oreilly, Sebastopol, California 1999

[3] *ARM, architecture reference manual.* Edited by Dave Jaggar, Prentice HallCambridge, UK 1996

[4] Rebecca Gurley Brace. *Intrusion Detection*. Macmilllan Technicall publishing, Indianapolis, IN

[5] Alessandro Rubini. *Linux Device Drivers*. Oreilly, Sevatopol CA, 1998

[6] D.P. Bovet, Marco Cesati. *Understanding the linux kernel*. Oreilly, Sebastopol CA, 2001

[7] Aurobindo Sundaram. *An Introduction to Intrusion detection systems.* ACM crossroads 1996

[8] B. Mukherjee, L. T. Herbelein and K.N Levitt. *Network Intrusion Detection*. IEEE Network, vol 8, no. 3, pp-26-41, May/June 1994

[9] Al Berg, *Knark & Dagger*. Information Security Magazine. April 2001

[10] Bruce Schneier and John Kelsey. *Secure Audit Logs to Support Computer Forensics*. ACM transactions on Information and System Security, Vol.2, No 2, May 1999, Pages 159-176

[11] Runar Jensen. *Malicious Linux Modules*, appeared in bugtraq. Oct 1997

[12] L. Van Doorn, G. Ballintijn and W. Arbaugh . *Signed Executables for Linux*

[13] G. H. Kim and E. H. Spafford. *The design and Implementation of TRIPWIRE: A File System integrity checker. Technical Report*, TR-93-071

[14] G. Kim and E. Spafford. Experiences with Tripwire, *Using Integrity Checkers for Intrusion Detection. In System Administration, Networking and Security*. Conference III. USENIX 1994.

[15] Halflife. *Bypassing Integrity Checking Systems*. In Phrack, volume 7, Issue 51 , September 1997

[16] S. M Beattie, Andrew P. Black, Cristing Cowan, Calton Pu, Lateef P. Yang. *Cryptomark: Locking the Stable door ahead of the Trojan Horse*. Technical review, Jul 2000

[17] H. K. Browne, W. A. Arbaugh, John Mc Hugh and William L. Fithen. *A trend Analysis of Exploitations*. In proceedings 2001 IEEE Symposium on Security and Privacy, pg 214-229

[18] Sandeep Kumar and Eugene H. Spafford. A pattern matching model for misuse intrusion detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11-21, October 1994.

[19] Dragon 5 Intrusion Detection Systems, Dragon Squire, Web ref: http://www.enterasys.com/ids/squire/

[20] R. Lehti, P. Virolainen, AIDE (Advanced Intrusion Detection Environment), Web ref: http://www.cs.tut.fi/~rammer/aide.html

[21] Cygnus, eCOS public-domain embedded RTOS. Web ref: http://sourceware.cygnus.com/ecos

[22] Embedded Linux Consortium, Embedded Linux, Web ref: http://www.embedded-linux.org/

[23] Microsoft Corporation, Window embedded. Web ref: http://www.microsoft.com/windows/embedded/

[24] Linux Kernel Archives, Linux Kernel source code, Web ref: http://www.kernel.org/

[25] R. King, ARM Linux port, Web ref http://www.arm.linux.org.uk/

[26] A. Rubini, Tour of the Linux Kernel Source, Web ref: http://www.linuxhq.com/guides/KHG/HyperNews/get/tour/tour.html

[27] R. Yahalom, B. Klein and T. Beth. *Trust relationships in secure systems---A distributed authentication perspective*. In Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy, pages 150--164, May 1993

[28] J. P. Anderson. *Computer security threat monitoring and surveillance*. Technical report, James P. Anderson Co., Fort Washington, PA, April 1980

[29] P. Marwick. *Vulnerability Assessment Framework 1.1,Critical Infrastructure Assurance Office*, October 1998

[30] Eugene H. Spafford. *The Internet Worm Incident.* Technical Report CSD-TR-933, September 19, 1991

[31] CERT® Advisory CA-2000-04 Love Letter Worm, May 2000

[32] CERT® Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL, August 23, 2001

[33] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222-232, February 1987.

[34] Katherin Price, Intrusion Detection Pages, COAST, **http://www.cerias.purdue.edu/coast/**, Sep 2000

[35] Mark Crosbie and Eugene Spafford. Active Defense of a Computer System using Autonomous Agents. In *Proceedings of the 18th National Information Systems Security Conference*, pages 549-558, October 1995.

[36] M. Ilgun, K. *USTAT - A Real-time Intrusion Detection System for UNIX*. Technical Report TRCS93-26, Computer Science Department, University of California at Santa Barbara, December 1993.

[37] Diego M. Zamboni. SAINT: A security analysis integration tool. In *Proceedings of the Systems Administration, Networking and Security Conference*, May 1996

[38] Smaha, S. E., Winslow, J. Misuse detection tools. *Computer Security Journal* 10(1994)1, Spring, pages 39-49.

[39] Vaccaro, H. S., Liepins, G. E. Detection of anomalous computer session activity. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 280-289, May 1989.

[40] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, M. Zissman. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. *In proccedings of the 2000 DARPA information survivability Conference and Exposition*, January 200

[41] R. Lippmann, D. Fried, J. Haines, J. Corba, and K. Das. Evaluating intrusion detection systems: Analysis and results of the 1999 darpa off-line intrusion detection evaluation. *In proccedings of the 3rd International Workshop on Recent Advances in Intrusion Detection (RAID 2000)*, October 2000

[42] CERT® Incident Note IN-2001-01, Widespread Compromises via "ramen" Toolkit, January 18, 2001

[43] Pedestal software. Integrity Protection Driver (IPD), http://pedestalsoftware.com/intact/ipd/

[45] M. Tanuan. *An Introduction to the Linux Operating System Architecture* **www.grad.math.uwaterloo.ca/~mctanuan/ cs746g/LinuxCA.html**

[46] A. Agnew. *DRAFT: SEBOS architecture*, Technical report, Missl lab September 2001

[47] Intel Corporation. *Datasheet, 21285 Core Logic SA-110 Microprocessor*, September 1998

[48] Intel Corporation. *Technical Reference manual, SA-110 Microprocessor*, September 1998

[49] Intel Corporation. *Reference Manual, strongARM EBSA-285 Evaluation Board*, October 1998

[50] Jesus Molina, Komoku Project webpage, http://www.missl.cs.umd.edu/komoku , MISSL Lab, Last Reviewed October 2001