

# TrustedBox: a Kernel-Level Integrity Checker

Pietro Iglio  
Fondazione Ugo Bordoni  
Rome, Italy  
e-mail: [iglio@fub.it](mailto:iglio@fub.it)

## Abstract

*There is a large number of situations in which computer security is unpopular. In fact, common users do not like too much restricted security policies. Usability is often preferred to security. Many users want to be free to use their computers to run untrusted applications. Moreover, it is not possible to require that every computer user is a security expert. As a consequence, it is very easy for hackers to gain access to a computer system, and to perform a number of unauthorized operations.*

*In this paper we focus on the problem of system integrity. There are some applications in which system integrity is at least as important as privacy and service availability. For this purpose, we have designed and implemented TrustedBox, a kernel-level integrity checker that can be used to enforce a very restricted security policy and that allows users to use the same system to perform untrusted operations.*

## 1 Introduction

Nowadays an increasing amount of critical tasks are carried out using computer systems. For this reason, computer security is, and will be, a primary goal in many modern commercial and governmental organizations. A lot of work has been done in recent years to protect systems against different kinds of attacks.

In spite of the progress in the field of computer security, there are some real life situations in which users are not protected enough when they carry out critical operations.

The main reason is that, in general, common users are not computer security experts. In most situations, they are not even computer experts. Such a lack of expertise is really helpful for hackers, because common users can hardly realize that the system has been compromised.

Moreover, a restricted security policy is often impractical because it would limit too much the usability of the system. In many organizations there is the need of using the same personal computer to edit documents, to

surf the Web, to exchange e-mail, to run multimedia applications, and to perform critical tasks.

For example, it is a common practice to use the same personal computer to run untrusted applications (such as applications downloaded from the network) and to buy products through sites offering e-commerce services. This common practice can be very dangerous, for several hacker's toolkits are widely available through the Internet. By using these toolkits, hackers could break into systems and replace critical operating system components to gain remote control of the system and to leave hidden backdoors for future access.

NetBus [NTB] and BackOrifice [BKO] are just two examples of software tools that can be used by hackers to have the complete control of the victim's personal computer. Such programs can infect any computer running Windows™ 95 and Windows™ NT by installing themselves as hidden system components. On many hacker Web sites it is explained how to use any executable program as a *trojan horse* to infect a Windows system using NetBus or BackOrifice. Some software tools are even publicly available to simplify this task. Once the system has been infected, hackers can remotely read all keystrokes, read and upload files, etc. on the infected machine.

Similarly, tools such as Rootkit [VEN] can be used to attack UNIX™ systems and to install modified versions of critical system components in such a way that it is very hard to detect their presence.

Companies pushing on e-commerce services advertise the absolute security of all communications with their servers because they are using strong encryption. Nevertheless, they never say that encryption is useless if the client has been compromised. Therefore, most users do not even know that their system could be remotely controlled by a hacker that, bypassing all cryptographic protections, could read all keystrokes, including passwords, credit card number, and so on.

In this work we focus on system integrity. There is a number of situations in which compromising critical system components, application or data can cause serious damages. For example, there are some countries in which the digital signature legislation states that digital signatures can be used to sign legal statements, such as contracts or commercial agreements. In these

countries, a compromised system running a modified version of the digital signature software run by a manager could lead to loss of money or other kinds of troubles. Similarly, a compromised Web server serving hacked pages could seriously damage the public image of a company.

In Section 2, we discuss the problem of protecting a user against a compromised system. First, we go through some typical approaches to the problem. Then, we explain why those approaches are still unsatisfactory to ensure a high-level of protection against experienced intruders. In Section 3, we present our solution. Details about the implementation are provided in Section 4. In Section 5 and Section 6, respectively, we show some examples of applications and we discuss some limitations of our solution. In Section 7, we compare our work with other existing solutions. Finally, we summarize our future plans and finish with our conclusions.

## 2 Is What You See What You Think It Is?

In the past years computer systems have constantly grown in complexity. About fifteen years ago personal computers were equipped with a very simple hardware and a primitive, single-task operating system that could run in a few kilobytes of memory. The number of users performing critical tasks by means of such systems was very low.

A modern personal computer is likely to have tens of applications installed, several gigabytes of hard disk storage, and hundreds of system components spread across the file system. Most operating systems are able to run more applications at the same time, and are based on an open architecture to be easily extended. Similarly, even the simplest application can have a very complex structure and be made of several components spread out in the system.

As a consequence, it is very hard to detect what is really going on when the system is running. From a user's point of view, the computing environments look simpler to use because of the progress of graphical user interfaces. The drawback is that a modern system is much more complex than those in use fifteen years ago. Of course, as complexity increases security is harder to enforce.

Even if a system is very carefully configured and a good security policy is somehow enforced, a single mistake made by an inexperienced user or, worse, by the system administrator could compromise the security of the whole system, or some parts of it. A single access to the system by an untrusted entity is sufficient to hide fraudulent code in the meanders of the system.

The basic problem is that, while much effort has been made to protect the system by unauthorized users, not too much care has been taken for the converse. Almost all modern operating systems provide authentication and access control mechanisms. However, to the best of our knowledge, there is no widely used operating system that provides a built-in mechanism for the user to authenticate the operating system itself, its critical components, and all running applications.

Common users, for example, are used to "authenticate" applications through their user interface. Nobody, of course, looks at the binary code of an application to check that the application itself is doing exactly what it has to do. How can users be sure that they are not performing critical operations on a compromised system?

Many solutions have been proposed to address this problem. A number of currently available tools, generally referred to as *audit tools*, fall into the following categories:

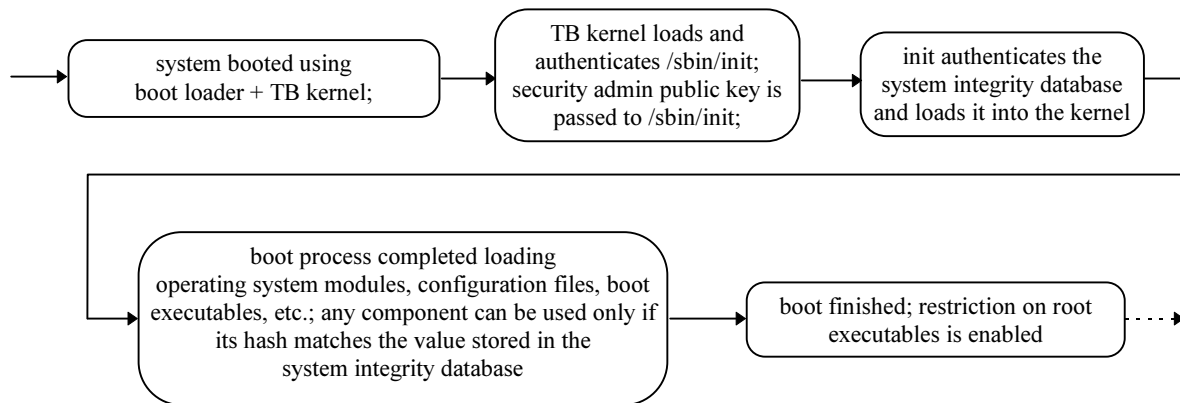
- **integrity checkers:** programs that compute a cryptographic signature for each critical system component or application, and compare the signature against a previously created database.
- **anti-virus programs:** programs that try to identify and block infecting programs by inspecting the content of files coming from an external source (e.g., a diskette or the network). They try to match the content of files with strings of bits from a database of known viruses.
- **static audit tools:** programs that check for system misconfiguration.

Note that some tools fall in more than one category. For example, some anti-virus programs include integrity checking capabilities.

We claim that none of the above approaches provides a satisfactory solution to the problem of protecting systems used to perform highly critical tasks.

A suitable configured integrity checking tool should always be used in a situation in which we want to be sure that a system configuration does not change over the time. Such tools can be used for monitoring the executable programs of the operating system, configuration files, and applications. The problem is that most of them do not perform continuous monitoring, but they have to be periodically run by the administrator. They can be useful to discover that a file has been modified, but they cannot prevent using modified files.

The limit of anti-virus programs is that they can successfully recognize only known computer viruses, or some known behavioral patterns (e.g., attempts to



**Figure 1:** TrustedBox kernel boot process

replace the boot sector). They are not guaranteed to work against new and still-not-classified viruses. Most of anti-virus programs, for example, can detect BackOrifice and NetBus, but they would probably fail to detect variants of those spy programs.

Similarly, static audit tools provide a valuable help to discover some known attacks, but they can do nothing if the attack has not been classified yet. Another problem is that many of them do not perform continuous monitoring, but they need to be run by the system administrator. Finally, we must say that such tools do not generally check whether an application has been compromised.

In addition to the above problems, most audit tools suffer from an inherent limitation: they rely on the underlying operating system. An experienced hacker could replace or intercept critical system calls to deceive an auditing tool. For example, it could be possible to change the *open* system call in such a way that when the auditing tool requests a compromised file, the system call returns the original version of the file. Even better, the compromised operating system could intercept all user requests for the execution of the audit tool and run a fake version of the audit tool itself.

It is possible to compromise the operating system in a number of ways. Once an intruder or a trojan horse has gained *superuser* privileges, there is nothing to do, even using active audit tools. Everything that runs before the audit tool is executed could attack the audit tool itself. For example, the system administrator has generally enough power to kill any process, and to replace any executable program. Thus, if we assume that no encrypted file system is used on the victim's computer, then an intruder that gains physical access to the system could install the hard drive or another system, and modify any audit tool by using a raw disk editing tool.

### 3 The TrustedBox Approach

The solution we describe is suitable for a large number of situations in which users do not need to perform unsafe and critical operations at the same time. Such situations are fairly common: some users use their personal computer most of the day to write documents and, occasionally, they have to issue digital signatures; a company Web server running is performing a critical task (i.e. serving "official" Web pages approved by the company itself), and sometimes the Web master has to update pages and perform administrative tasks.

Under these circumstances, the TrustedBox approach has the following advantages:

- users can be effectively protected against highly motivated individuals that could somehow replace any file in the file system;
- users can use the same system, at different times, both for unsafe operations and highly critical tasks;
- the role of the security administrator is separated from the role of the system administrator; the system administrator does not even need to be a trusted entity;
- it is possible to enforce a restricted security policy for critical tasks;
- there is no need to modify existing applications to be aware of new security mechanisms;
- there is no need to keep a system used for critical tasks in a secure location with restricted access;
- it is inexpensive and practical;

We distinguish two security policies, a *relaxed policy* and a *restricted policy*, corresponding to two system states, the *untrusted state* and the *trusted state*.

When in untrusted state, the security policy could be relaxed in favor of usability. Users could have permission to download and run unsafe applications from the network, to enable dangerous network services, to allow system access to other users, and so on. In real life, it is very common that users have such requirements for their activity.

It is obvious, however, that a system running in untrusted state can be easily compromised. For this reason, users are required to switch to the trusted state when they have to perform critical tasks. To enter in trusted mode the user has to shutdown the system and boot with a *TrustedBox kernel* (TB kernel). This is necessary because any part of the system could have been modified, including the operating system itself.

A TB kernel is based on a regular kernel that has been modified to enforce the restricted policy. Our prototype is based on a UNIX kernel, but most concepts apply to other operating systems as well.

The TrustedBox kernel has been previously stored on bootable media (e.g. a diskette) that must be kept from the user in a safe place. The TB kernel contains a minimal kernel required to boot the system and access the file system. Note that the integrity of the whole system relies on the integrity of the boot media.

Once the kernel has been loaded, it verifies the integrity of the `/sbin/init` executable. Under the UNIX operating system, `/sbin/init` is the first program that is run at boot-time by the kernel, and it is responsible to complete the boot process. The integrity of `/sbin/init` is verified comparing its message-digest with the corresponding message-digest stored on the boot media.

When `/sbin/init` is launched, the TB kernel passes the public key of the security administrator as a command line option:

```
/sbin/init -pk public-key
```

Note that the public key is stored on the boot media, therefore it comes from a trusted source. The modified version of `init` uses the public-key to authenticate the *system integrity database*, a database containing the message-digest of all critical files, including executables, kernel modules, configuration files, etc.

Next, `init` loads the system integrity database into the kernel and continues a normal boot process. Kernel modules can be loaded, services started, and so on. However, since we are starting a trusted system that is stored on an untrusted file system (except the boot kernel), every single component is authenticated using the message-digest from the system integrity database. If the message-digest for a file does not match the value

stored in the system database, the kernel refuses to open the file.

The diagram in figure 1 summarizes the trusted boot process. In the rest of this section we describe the restricted security policy that can be enforced using the TB kernel, plus some other details.

### 3.1 Security Policy in Trusted Mode

The TB kernel distinguishes between *trusted files* (i.e. files listed in the system integrity database) and *untrusted files* (the remaining files). In addition to the standard security mechanisms of the underlying operating system, the TB kernel imposes a number of restrictions concerning program execution and file system access. In particular, an application can be executed if and only if:

- the user is authorized to run the application, according to the standard security mechanisms of the underlying operating system;
- the application is listed in the system integrity database;
- the application message-digest matches the message-digest stored in the system integrity database;

Furthermore, an application can be executed with *superuser* privileges if and only if it meets the above requirements and is marked with the *super* option in the system integrity database. This restriction has been added because applications running with *superuser* privileges are so powerful that can bypass many security mechanisms, especially on UNIX systems. The security administrator can limit to the minimum the set of *superuser* applications that can be executed in trusted mode setting the *super* option only for the applications required to perform the critical tasks. It is possible, for example, to not permit the execution of any shell with root permission. Note that this is a strong restriction to the common operating system policy, where the *superuser* can execute every application.

Since some applications may need to run with root privileges during the boot process (e.g. `/bin/sh` can be needed to start some daemons), the restriction on the execution of *superuser* applications must be explicitly enabled using a new system call, *tbox\_set\_state*. Such system call is used to raise the TB state from *booting*, that is the initial state, to *running*, that is the normal state in which the user works.

A correctly configured system should raise the execution state to *running* once the boot process has been completed to enable the restriction on executables that run with root permission.

Note that the system call has been implemented in such a manner that, once the state has been raised to *running*, it cannot be changed for the rest of the session.

As far as non-executable files are concerned, the policy is slightly different. The TB kernel authorizes a regular process to open a file with *read* permission if and only if:

- the file does not appear in the system integrity database (i.e. it is an *untrusted file*);  
*or*
- the file appears in the system integrity database and its message-digest matches the digest in the configuration file;

Permission to open untrusted files is granted so that it is possible to run a large number of applications that work on newly generated files or access to files modified during the trusted session. For example, a linker needs to access files created by a compiler. Since such files have been created during the current trusted session, they are not stored in the system integrity database (that must exist at boot time). Note that, unlike regular files, executable files can be only executed if they exist at boot time.

The authorization to open untrusted files can lead to security problems in some circumstances. For example, if our application needs a *perl* interpreter to run, once we must authorize the execution of the *perl* interpreter in trusted mode. In this way, it would be possible to execute any *perl* script, because the execution of a script, from a kernel point of view, corresponds to an “open” operation (and not to an “execution” operation). However, we want to have a strict control on the set of programs that can be executed (regardless of the fact that they are binary programs or scripts). To address this issue, an entry in the system integrity database for an executable program can be marked with a special flag, *open\_only\_trusted*, and the following rule holds:

- the TB kernel authorizes a process marked with a *open\_only\_trusted* flag to open a file for reading if and only if the file appears in the system integrity database and its message-digest matches the digest in the configuration file;

The last constraint allows the security administrator to safely authorize the execution of interpreters, because it enormously reduces the risk that users run *trojan horses* hidden inside untrusted scripts.

It is easy to understand that the TB kernel strongly relies on the integrity of any executable, library,

configuration files etc. that is used when the system is in trusted state. For this reason, the TB kernel rule imposes this additional restriction:

- it is not possible to change any file, executable or not, that appears in the system integrity database;

As far as directories are concerned, it is possible to sign the content of a directory, i.e. to sign the listing of a directory at a given time. A directory listed in the system integrity database is said to be a *trusted directory*. Here are the limitations imposed by the TB kernel for trusted directories:

- it is not possible to change the content of a trusted directory (e.g. adding or removing files from that directory);
- it is not possible to open for listing a trusted directory whose signature does not match the value stored in the system integrity database;
- a trusted directory can only contain trusted files and other trusted directories;

The signature of a directory is useful to avoid any change to the boot process. Under many UNIX systems, in fact, the boot process is carried out running all programs belonging to some special directories (e.g. */etc/rc.d*). The security administrator can “sign” a particular boot process signing these directories and all contained executables.

Finally, for any operation not mentioned here, the normal access control policy of the operating system is applied.

## 4 Implementation Details

As a proof of concept, we have developed an experimental prototype of the TB kernel based on a modified version of the Linux kernel. The prototype is not meant to be a complete and highly-secure implementation, but it is useful to evaluate the usability of a system running in trusted state. At the time of writing, the prototype does not implement integrity checking for directories and other minor details.

A substantial work has been done on the Linux kernel to change the semantics of some system calls (*open* and *execve*) and to add new system calls. The changes to the Linux kernel can be summarized as follows:

- an implementation of the SHA algorithm has been added at kernel level;

```
fb0347750f82b8bfff5684f11fc586ebddd919878 /sbin/init
b1b67ad426bc408d68cb817aece75effde1533b3 /etc/syslog.conf
3dd6a70d8bec298033808cf9647c664043a7e91e /etc/rc.d/
71c31f59ae2896a3ff6ca6e7092eece0f2db2bf3 /bin/ping super
ec636b4c89a82b5c239e3bc8cf8ccce6c4d19e042 /mnt/nfs/data always
a09d4e2eef5b4014d8987fee1696749556e04e8c /usr/bin/perl super, open_only_trusted
```

**Figure 2:** Example of system integrity database entries

- a new boot-time parameter has been added; the new parameter is used to pass the message-digest of the system integrity database to the kernel;
- the *open* and the *execve* system calls have been modified to implement the policy described in the previous section;
- a system call, *add\_sid\_entry*, has been added; this system call is used by the *trusted\_init* program to load the system integrity database into the TB kernel;
- a new system call, *tbox\_set\_state*, has been added to enable the restriction on root executables;

Furthermore, we have obtained the *trusted-init* program modifying the normal *init* program to accept an extra parameter (the security administrator public key), authenticate the system integrity database and to load it into the kernel. In particular:

- the TB kernel runs *trusted-init* passing the public key stored on the trusted boot media;
- *trusted-init* passes the public key to a cryptographic module that verifies the signature of the system integrity database. The cryptographic module is statically linked, to avoid trusting external libraries or programs;
- *trusted-init* repeatedly invokes the *add\_sid\_entry* system call to pass each entry to the TB kernel;
- after that the last entry has been passed to the TB kernel, *trusted-init* calls *add\_sid\_entry* passing a special parameter that stops the kernel from accepting further entries. In this way, it is not possible for other programs to add new entries that are not stored in the system integrity database;
- *trusted-init* completes the normal boot process;
- after the boot process has been completed, the following invocation:

```
tbox_set_state(TBOX_STATE_RUNNING);
```

enables the restriction to run as root only executables that have been marked with the *super* flag.

The signature of the system integrity database is verified using a publicly available implementation of the RSA algorithm [RSA] using a MD5 hash algorithm. Of course, any other algorithm can be used with minimal changes, according to the security requirements of a particular application.

The boot diskette is prepared copying on it the TB kernel and the Linux Loader (LILO). Since the security administrator public key is too long to be passed as a kernel parameter, it has been hard-coded into the TB kernel. Currently, this requires that the kernel is partially re-compiled when the public key is changed. Changing the public key, however, should not happen frequently. However, we have planned the development of maintenance tools to simplify the process of storing the public key on the boot media.

The system integrity database is stored into a text file, */etc/sysintdb*. A system integrity database entry has the following format:

```
hash filepath [super][always][open_only_trusted]
```

The *hash* value is an ASCII representation of the message-digest for *filepath*. The optional *super* keyword means that the file can be executed with root privileges. This keyword is meaningless if the file has no execute permission. The *always* keyword can be optionally specified to force integrity verification every time the file is open. This keyword is related to an optimization that is explained in the subsection “Performance Improvements”. Finally, the *open\_only\_trusted* keyword is used to specify that the executable file can access only files listed in the system integrity database.

Some sample entries are shown in figure 2. The cryptographic signature of the system integrity database is stored into the */etc/sysintdb.sig* file.

#### 4.1 Performance Improvements

Invoking a message-digest calculation for each *open* and *execve* system call invocation introduces a performance overhead, since these system calls are used

very frequently. In order to greatly reduce this overhead, we have adopted the following optimization: each file to be authenticated is authenticated only the first time it is accessed. Then the file is marked, for the rest of the trusted session, as *authenticated*. No other checks will be performed after that the file is marked as *authenticated* until the next system shutdown.

Note that, since it is not possible to open for changes a file listed in the system integrity database, the authenticated file will not change its contents for the whole session. However, it is possible to disable this optimization for some files specifying the *always* keyword in the system integrity database. In such case, the file will be checked at each access.

Further performance improvements could be gained using a faster implementation of the SHA algorithm, or a faster message-digest algorithm. Performance, however, was a secondary issue respect to security for the kinds of applications we had in mind during the design of TrustedBox.

## 5 Examples of Applications

As explained in the previous sections, TrustedBox provides a valuable help to run applications in a trusted environment without using dedicated machines with a very restricted and controlled access. Once the system has been correctly configured, it can be used even by inexperienced users without too many restrictions. Their only responsibility is to perform highly critical tasks in trusted mode and to keep the boot media in a secure place.

An example of application in which TrustedBox can be really helpful to solve a lot of security and administrative issues is the generation of digital signatures. We can imagine the following scenario: a number of users in the same organization use digital signatures to sign critical documents, such as financial transactions or contracts. The same users, however, use their personal computers for other tasks, such as word-processing, e-mail exchange and Internet surfing.

The scenario described is very common, as digital signature is becoming very popular in many modern organizations. A security administrator is responsible to set up, on a isolated machine (disconnected from the network and running in single-user mode), a safe environment with the minimum set of applications required for the trusted mode. The safe environment can include, for example, a document viewer (such as a HTML browser), a smart-card driver and a software to digitally sign documents.

After that all the software has been carefully verified, the security administrator can use the appropriate maintenance tools to create the system

integrity database and the boot diskettes that will be distributed to all employees.

The system administrator can then install the trusted configuration, along with the system integrity database, and all network connected machines that need to run the trusted environment. As previously pointed out, the security administrator and the system administrator can be two different entities, and the system administrator could even be untrusted. Once users have got their boot media, in fact, the integrity of the subsystem used for critical operations will be automatically checked.

There is no way the system administrator can install a modified configuration (assuming that the safe configuration has been correctly chosen, of course).

Users can now perform their regular tasks booting in untrusted mode: for example, they can edit documents and download documents to be signed from the network. Periodically, e.g. twice a day, they boot in trusted mode, view the documents with the authenticated viewer, and sign them using a smart-card.

Note that in a similar scenario the responsibility of the system administrator is greatly reduced. He must not keep under constant monitoring a large number of critical workstations and, on the other hand, users don't have to trust the system administrator when they sign critical documents. As far as users preserve the integrity of the boot media, there is no need to prevent physical access to their workstations when they are not running in trusted mode.

Another situation in which TrustedBox can be worth using is for running specialized servers, i.e. servers running a restricted number of applications. Examples are critical Web servers or firewalls. Running a system like a Web server in trusted mode could greatly limit the potential damage by an intruder that gains unauthorized access to the system. In the past months, for example, a number of Web sites have been hacked. A typical attack consists of a replacement of some Web pages with a modified version of the same pages. This kind of attack can seriously damage the credibility of a large organization.

If a TB kernel is used to run the Web server and the Web pages are added to the system integrity database, it would be virtually impossible to bypass the kernel-level imposed restrictions to modify trusted files.

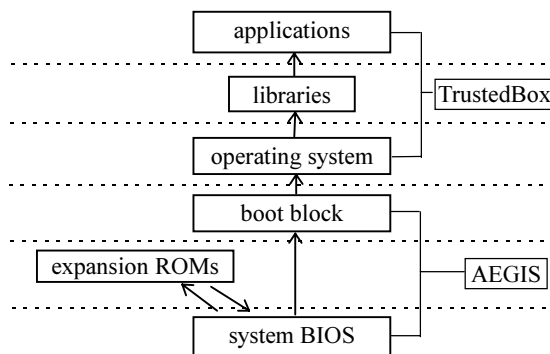
Likewise, it would be really hard to modify the configuration of a firewall that runs on the top of a TB kernel. A security company could install and configure firewalls whose configuration cannot be modified even by the customer. In this way, it should be possible to avoid a potential dispute about a firewall misconfiguration, since the firewall could only run with the provided configuration.

## 6 Limitations of TrustedBox

Despite the valuable help of TrustedBox in the protection of system integrity, it has some inherent limitations.

From a security point of view, one of the most fundamental limitations is that TrustedBox does not authenticate the hardware and the firmware. In particular the software that is executed when the system starts up and that is responsible for starting the boot process (called BIOS on Intel-based machines), cannot be authenticated, as it is loaded before the kernel.

If the hardware or the firmware is compromised, in fact, the user cannot be sure that the kernel that is loaded upon start up is the trusted kernel that is stored on the boot diskette. Latest personal computers have a software upgradable BIOS, and several computer viruses do significant damage overwriting the system BIOS. Thus, a possible attack to the TrustedBox system could be done replacing the system BIOS with a modified version that, for example, performs a different bootstrap process. Since the authentication starts only after that the kernel has been loaded, there is no authentication of the underlying layers.



**Figure 3:** TrustedBox and the AEGIS architecture

This problem is discussed and addressed in [ARB], that proposes a secure bootstrap architecture. The secure bootstrap process, called AEGIS, ensures the integrity of the bootstrap code computing a cryptographic hash for the system BIOS, any other firmware (eg. graphic card and disk controllers) and the boot sector. The hash values are compared with a signature associated with each component and stored on a read-only memory. AEGIS relies on a trusted system ROM, containing the verification code and the cryptographic certificates. A basic assumption, of course, is that no unauthorized user can access the system motherboard and replace the ROM.

Since AEGIS addresses the problem of validating any layer that is not checked by TrustedBox, it is possible to say that using both of them on the same system would be a great security enhancement. In this scenario, AEGIS would be responsible for ensuring the integrity of the firmware bootstrap process, and TrustedBox would ensure the integrity of the operating system during the rest of the boot process and when the system is running, as shown in figure 3.

Another security problem with TrustedBox is that it can be subject to denial-of-service attacks. If an intruder changes or removes any signed executable file, for example, the system will not execute that file in trusted mode. However, this is a minor problem if compared to the damage of running a compromised executable during critical tasks. The system administrator should keep a backup copy of any signed file to quickly recover a compromised system. Alternatively, it could be possible to use same solution discussed for the AEGIS architecture in [ARB2]. This solution is based on a trusted repository, that can either be an expansion ROM board that contains verified copies of the required software, or it can be a network host. In the case of TrustedBox, the network host solution could be used, using a protocol with strong authentication to ensure the integrity of transferred data.

Another possible attack to a TrustedBox system could be performed modifying the warm-reboot procedure of the operating system in untrusted state. In fact, a compromised operating system could “pretend” that is doing a warm-reboot and that it is loading the TB kernel. As a consequence, the user could erroneously believe that the system is in trusted state and could perform critical tasks while the compromised system is running. For this reason, we strongly recommend that the trusted mode boot should be only performed after a complete system power-off. Unfortunately, we do not know any way to impose that users enter in trusted mode after a cold-boot.

Finally, there is a potential security problem due to the need to allow users booting from an external media. In many situations users are not permitted to boot a system from a diskette, as they could boot with an operating system that allows to access the file system bypassing all security checks. Since we cannot impose this restriction, one possible solution is to use TrustedBox in combination with a cryptographic file system, such as [CFS], or using a special boot device, as explained in the Section “Future Works”. Another possible solution is the usage of a modified system BIOS that authenticates the boot media, so that it is not possible to bootstrap with a different operating system.

From a practical point of view, a major problem is the need to reboot the system to enter in trusted mode.

This could be a serious problem if the activity requires a frequent switch between critical tasks and non-critical tasks. This is an inherent limitation of our approach, that does not fit very well in such situations.

It could be possible to set up a configuration file to run as many applications as possible in trusted mode, but there are some tasks that cannot be performed with such a restricted security policy. One example is software development: it is not possible to compile and run applications in trusted mode, because any binary generated during the trusted session cannot appear in the configuration file and, thus, cannot be executed. A trusted session, however, could be used to compile secure applications or set up safe scripts without running them.

## 7 Existing Solutions

Many available security tools try to address the problem of system integrity. One of the most popular in the UNIX world is probably Tripwire [KIM]. Tripwire helps the system administrator to detect file tampering by comparing the cryptographic signature of each protected file against a signature stored in a database. Several signature algorithms can be chosen, including MD5 [MD5], MD4 [MD4] and SHA [SHA]. To prevent the database from being altered, the authors suggest to store it on some tamper-proof media. Similar tools are ATP [ATP], that employs a dual signature (32-bit CRC and MD5) to verify files, and Binaudit [BSP].

COPS is an intrusion detection tool that checks a UNIX host for well known vulnerabilities in the system set up. It focuses on things like inappropriate permissions set on files and directories, missing or inadequate password security, potentially dangerous *setuid* programs and similar problems. After its release, COPS has been extended to include an integrity checker, the *crc\_check* program, based on a simple CRC checksum of the files being monitored.

As previously pointed out, a main disadvantage of such tools is that they rely on the integrity of the operating system and other critical programs. Even if such tools are stored on a read-only media, it is very simple, for example, to modify the source code of any public domain shell to trap any invocation of `/usr/bin/cops` or `/usr/bin/tripwire` and execute a tampered version of such tools. Another disadvantage is that traditional tools have to be run periodically and they cannot guarantee a constant integrity of the system. Therefore, such tools are a valuable help to detect tampering after it happens, rather than preventing it. As explained in the previous sections, the TrustedBox approach provides a constant monitoring of the system during the execution in trusted mode.

Domain and Type Enforcement (DTE) [WAL] is an access control technology that aims to minimize the damage root programs can cause if subverted. An operating system providing DTE can be configured to enforce a restricted security policy to confine root programs in domains from which they can access only a subset of system objects (files, devices, etc.). Some implementations of DTE are available, for example as extensions of a UNIX kernel. A drawback of the DTE approach is that, to be effective, some system applications should be modified to fully take advantage of DTE mechanisms.

Since DTE does not address the problem of system integrity, its approach is complementary to the TrustedBox approach. An implementation of DTE in a TB kernel could be useful to better define a restricted security policy required to carry out critical tasks.

An example of security application that is based on two different kernels is Sidewinder™ [THO]. Sidewinder™ is an Internet firewall based on a modified version of BDS-OS UNIX that includes DTE. It uses an operational kernel, with DTE enabled, and an administrative kernel, in which security policy checks are bypassed. System maintenance tasks are performed under the administrative kernel, that can be started by a user that is “physically” connected to the system and only after shutting down the operational kernel.

The difference between the Sidewinder and the TrustedBox approach is that the Sidewinder system administrator has to be trusted. This because any software modified or installed by the system administrator working in administrative mode will be executed in operational mode. Using TrustedBox, instead, nobody but the security administrator can alter the configuration that will be used in trusted mode. Only the security administrator can authorize changes to the system issuing a digital signature for the system integrity database. The Sidewinder approach can be good for a firewall, that is normally not accessed by untrusted users, whereas our approach is suitable for systems that are normally used to run untrusted applications.

## 8 Future Works

We are actively working on our prototype of the TrustedBox kernel to fully implement all discussed issues. Furthermore, we aim to provide administrative tools to simplify the task of the security administrator, to update the system integrity database, to create the trusted boot diskette and to set up an appropriate security policy.

An interesting future direction would be the integration of the TrustedBox approach with other kernel-level solutions to improve UNIX security, such

as DTE and capabilities [MOR]. Also, we are planning the integration of TrustedBox with the AEGIS architecture, discussed in section 6.

Finally, we are investigating the possibility to use special boot devices, such as PCMCIA cards, to both authenticate the user that performs the system boot and to reduce the risk that the boot media is tampered. The firmware should authenticate the PCMCIA card, load the TB kernel from the PCMCIA card and start a regular boot process. The PCMCIA card would also be safer than a traditional diskette to store the trusted boot media, but with the drawback of additional hardware costs.

## 9 Conclusions

We have designed and implemented a secure operating environment that can run on the top of an untrusted file system to perform highly critical tasks in a safe manner.

Thanks to its mechanisms to define a restricted security policy, a well configured system running in trusted mode is much harder to attack compared to a conventional system, especially for intruders that aim at modifying the system configuration or replacing system components. On the other hand, the only user responsibility is to start the system with a trusted boot media.

If the security administrator and the system administrator are distinct entities, furthermore, users do not have to trust the system administrator, as normally happens.

The ability to boot in untrusted mode or in trusted mode, so that the same system can be used both for unsafe and secure operations, makes TrustedBox a very interesting and inexpensive solution for a large number of application fields.

Digital signature systems and secure servers are just two cases in which TrustedBox could be adopted, but it would be very easy to think about a number of similar situations.

## References

- [WAL] Walker K. M. et al., "Confining Root Programs with Domain Type Enforcement (DTE)", in Proceedings of Sixth USENIX Security Symposium, San Jose (California), 1996.
- [KIM] Kim G. H., Spafford E. H., "The Design and Implementation of Tripwire: a File System Integrity Checker", in Proceedings of 2<sup>nd</sup> ACM Conference on Computer and Communications Security, 1994.
- [BDG] Badger L. et al, "A Domain and Type Enforcement UNIX Prototype", USENIX Computing Systems, Vol 9, No. 1, 1996.
- [THO] Thomsen D. J., "Sidewinter: Combining Type Enforcement and UNIX", Proc. 11<sup>th</sup> Computer Security Application Conference, Orlando (FL), 1995.
- [VEN] Venema W., "Root Kit", Presentation at SURFnet CERT-NL SGC-SEC/SSC Workshop, May 1995.
- [FAR] Farmer D., "The COPS Security Checker System", Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, p. 165.
- [ATP] D. Vincenzetti, M. Cotrozzi, "ATP anti tampering program", proceedings of the Security IV Conference, Berkeley, CA, 1993. USENIX Association.
- [BSP] M. Bishop, "Auditing files on a network of machines", Security Workshop, USENIX, 1988.
- [ARB] Arbaugh W. A., Farber D. J., Smith J. M., "A Secure and Reliable Bootstrap Architecture", in Proceedings of IEEE Symposium on Security and Privacy, Oakland (California), 1997.
- [ARB2] Arbaugh W. A., Farber D. J., Smith J. M., "Automated Recovery in a Secure Bootstrap Process", Internet Society Symposium on Network and Distributed System Security (SNDSS) in San Diego, March 11-13, 1998.
- [SHA] National Institute for Standards and Technology, "Secure Hash Standard", Federal Information Processing Standards Publication 180, Government Printing Office, Washington, D.C., 1993.
- [MD4] Rivest R. L., "The MD4 message digest algorithm", Advances in Cryptology – Crypto '90, 1991.
- [MD5] Rivest R. L., "RFC 1321: The MD5 message-digest algorithm", technical report, Internet Activities Board, April 1992.
- [RSA] R. L. Rivest, A. Shamir and L. Adelman, "A method for obtaining digital signatures and public-key cryptosystem", Commun. Of ACM, Vol. 21, No. 2, pp. 120-126, Feb. 1978.
- [CFS] M. Blaze. "A Cryptographic File System for UNIX", proc. 1<sup>st</sup> ACM Conference on Computer and Communications Security, Fairfax, VA, November 1993.
- [MOR] A. G. Morgan, "Linux-Privs", <http://www.kernel.org/pub/linux/libs/security/linux-privs/doc/linux-privs.html/linux-privs.html>
- [NTB] NetBus: a remote administration and spy tool, <http://www.netbus.org>
- [BKO] Back Orifice Windows Remote Administration Tool, <http://www.cultdeadcow.com/tools/bo.html>