

A Trusted Linux Client (TLC)

David Safford, safford@us.ibm.com
Mimi Zohar, zohar@us.ibm.com
T.J. Watson Research Center
IBM

Abstract:

The goal of the Trusted Linux Client (TLC) project is to protect desktop and mobile Linux clients from on-line and off-line integrity attacks, while remaining transparent to the end user. This is accomplished with a combination of a Trusted Computing Group Trusted Platform Module (TPM) security chip, verification of extensible trust characteristics, including digital signatures, for all files, authenticated extended attributes for trusted storage of the resultant file security meta data, and a simple integrity oriented Mandatory Access Control (MAC) enforcement module. The resultant system defends against a wide range of attacks, with low performance overhead, and with high transparency to the end user.

Introduction

Windows client machines have been the target of ever increasing attacks, such as viruses, adware, spyware, browser exploits, and spam containing malware attachments. Recent studies showed that 80% of all windows clients had spyware installed[1], and 30% had been compromised with back doors[11]. While Windows machines are subject to many Windows specific attacks, they are also subject to operating system independent attacks which could easily be mounted against Linux clients. These attacks include email attached malware, browser downloaded malware, and phishing emails which mount social engineering attacks on user's sensitive account data. Linux client users need strong protection for their system software and authentication keys to defend against these potential, and likely coming, attacks.

It is critical, however, that this protection be transparent to the user. Typical client users are unable to diagnose security issues, and cannot be relied upon to understand

or respond to security mechanisms which interfere with normal operation. TLC must not startle the user, but must, as quietly as possible, protect the system from harm. TLC also must not visibly impact performance, or users will not tolerate it.

In the past, the techniques needed to protect against integrity attacks were difficult to implement, difficult to use, and imposed a high computational performance penalty on normal operation. Two technologies have become available to help overcome these past problems: the Linux Security Module (LSM) framework in Linux kernel 2.6, and the Trusted Platform Module (TPM), which is now available on most client computers, including those from IBM, HP, Fujitsu, and Dell.

LSM provides all the necessary hooks throughout the Linux kernel to enable a kernel module to mediate all security sensitive decisions from a single loadable module. In the past a security implementor would have to patch large numbers of kernel files to mediate all the needed security decisions. Keeping such patches current against a continually changing code base was simply not practical. With LSM, the necessary hooks are now available in all the necessary locations, so that a security designer can take advantage of them with little effort.

The TPM chip provides several essential hardware based security services. First, it provides hardware based public key management and authentication services whose private keys cannot be stolen by software based attacks of any kind, (including malware and phishing attacks), as the sensitive private keys are generated on the chip, and are never visible in plain text outside of the chip. Second, it provides the essential boot-time hardware to measure a system's software integrity, and report any sign of software tampering. Third it provides for secure storage of data, including keys, protecting their secrecy across reboots and protecting them against

theft. These integrity measurement and secure storage functions enable significant performance improvement in the verification of file authenticity, integrity, currency, and safety, so that these functions can be performed efficiently.

Using the LSM and TPM systems, TLC implements three complementary loadable kernel modules for protecting client integrity:

- Trusted Platform Module (TPM)
- Extended Verification Module (EVM)
- Simple Linux Integrity Module (SLIM)

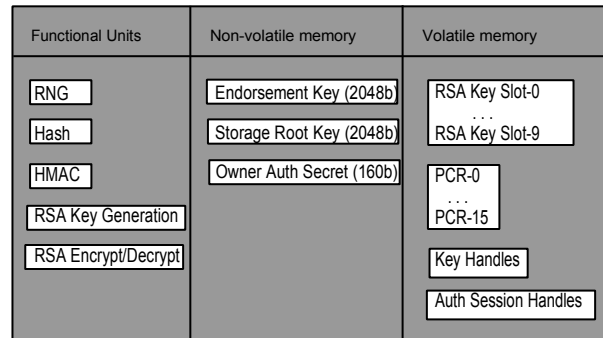
TLC has been implemented and tested on Fedora Core 3 and Red Hat Enterprise version 4 systems, and the following descriptions have details, such as RPM package management, which are specific to those distributions. The TLC kernel modules, however, are as generic as possible, so the porting the support applications to other distributions should be simple.

TPM – Trusted Platform Module

The Trusted Computing Group [13], has defined an open specification for a TPM, which has been implemented by multiple chip vendors, and incorporated into desktop and mobile systems from the major PC manufacturers. Here is a picture of IBM's LPC bus version, based on an ATMEL TPM chip:



While the full TPM specification is quite long and difficult to understand, the chip's basic functionality is simple. Linux Journal published an introduction to the chip in [10], which also described an open source package [9], which enables Linux users to test and develop applications. From a programmer's perspective, a TPM looks like the following logical diagram.



The chip has a hardware random number generator (RNG) and RSA engine for on-chip key pair generation. When a key pair is generated, the private part is encrypted by the Storage Root Key (SRK) or a descendant, and the resultant pair exported out of the chip for storage. The chip has 10 volatile slots into which the key pairs can be loaded, decrypted, and then used for signature, encryption, or decryption. (Signature verification is not done on-chip, as it is not a sensitive operation.)

The TPM chip also has sixteen Platform Configuration Registers (PCR), which are used to securely store 160 bit hashes. These hash registers are used to store hashes of the software boot chain (BIOS, master boot record, grub bootstrap loader, Linux kernel, and initial ramdisk image). Then the usage of keys for encrypting or decrypting can be tied to specific values of these PCR registers, so that if any part of the measured software is altered, the decryption is blocked. In TPM terminology, encryption tied to a specific PCR value is called “sealing”, and the corresponding decryption called “unsealing”. Malicious alterations to the master boot record, grub, kernel, or initrd cannot escape detection through the PCR values, as the measurements are always done on the next boot stage, before execution is transferred to it. Since the TPM hashes all presented data into a given PCR, it is computationally infeasible for malicious code to calculate and submit a measurement which would result in a target “correct” value. after this hashing.

TLC uses the TPM to create and seal a 160 bit random kernel master key. This sealed kernel master key is

unsealed by the TPM and released to the kernel at boot time, if and only if the following conditions are true:

- the PCR values have not changed
- the sealed kernel master key is provided
- the user provides the unsealing password

TLC supports storing the sealed kernel master key in the init ramdisk, or on a USB flash drive. If the sealed kernel master key is stored on a USB flash drive, and a notebook, such as an IBM T42p with fingerprint reader is used, then the boot success is based on

- “what you are” (fingerprint),
- “what you have” (USB token),
- “what you know” (sealing password), and
- the measured integrity of the boot sequence, up through the initial ramdisk.

TLC supports having a single sealed kernel master key across all users, in which case an administrator would have this sealed key and authorization password, and would present these at boot time, and users would log in normally after boot. TLC also supports a model in which each user has their own unique copy of the sealed key, with unique authorization password. In this case when they authenticate to the TPM at boot time, this also serves to authenticate them for login purposes, and the GDM graphical login program treats them as preauthenticated.

There are some interesting implementation challenges for this boot-time unsealing of the kernel master key. The natural time to load the kernel master key, and the TPM, EVM, and SLIM modules is just after grub has measured the kernel and init ramdisk, but before any files on the root file system have been accessed. This means that the unsealing has to occur during the running of the init ramdisk's “init” nash script, and all necessary files and programs must exist in the init ramdisk. This environment is very primitive – all executables must be statically linked, and must be small. The normal TPM software stack (TSS) is not available, so a special lite version of libtpm [9] was used, compiled with the diet libc package. This reduced the program for loading the

kernel key from an original 2MB (including libraries), to 25KB.

A second challenge was that the normal TPM device driver sends a sealed blob to the TPM, and returns the unsealed data back to the application, which is not desired when loading the kernel master key. To keep the kernel master key secret from user space, a special pseudo TPM command (TPM_UnsealKernKey) was defined. The driver was modified to recognize the pseudo unseal command, and to not return the unsealed data, but rather to store it as the kernel master key. Similarly, the sealed kernel key is initially created at install time with a pseudo TPM command, TPM_SealKernKey, which takes random bits from the TPM hardware random generator, has the TPM seal the data, and returns only the sealed kernel key to user space. In this way, the kernel master key is never visible outside of the TPM kernel module. The TPM based kernel master key is then hashed to derive subsequent keys for use by other kernel modules, such as EVM, and the encrypted loopback. Even if one of these modules leaks their key information, the master key, and other derived keys are not compromised.

While the TPM could be used to measure all files, even after boot, it is not a fast chip, so the verified kernel, with EVM, is used to do software integrity verification of all subsequent files.

EVM - Extended Verification Module:

For security reasons, it is desirable to check security characteristics, including the authenticity, integrity, revision level, and robustness of an application before its execution, to determine whether or not to run the executable, or under what level of privilege to run it. Mechanisms such as Message Authentication Codes, signed executables, anti-virus, and patch management systems have been implemented to address one or more of these issues. (For clarity, Message Authentication Code will be abbreviated HMAC, and Mandatory Access Control abbreviated MAC to differentiate them throughout the paper.) EVM presents a single comprehensive mechanism, Extended Verification, to cover all of these goals, but implemented in a single,

optimally fast mechanism, with a flexible policy based management system.

The use of signed executables to prevent the execution of malicious programs, such as viruses, was first described by Pozzo and Gray in [8]. This paper proposed that executables be digitally signed, and that the kernel check the signature every time an executable is to be run, refusing to run it if the signature is not valid. Viruses or other malicious codes, lacking a valid signature would be unable to run.

The digital signatures on the executables can be of two types: symmetric or asymmetric. A symmetric signature uses a secret key to key a Message Authentication Code (HMAC), taken across the entire content of the executable file. Symmetric signatures can be verified with relatively little overhead, but the key must remain secret, or the attacker can forge valid signatures. This makes symmetric signatures useful mainly in the local case. In addition, the key must be kept secret on the local machine, and this is very difficult to do.

An asymmetric signature uses a public key signature pair, such as with the RSA signature scheme. In this case the private key is used to sign, and the public key is used to verify the signature. The private key needs to remain secret, but need exist only on the signing system. All other systems can verify the signature knowing only the public key, which need not be secret. Thus executables signed with asymmetric signatures are much more flexible, as the signed executable can be widely distributed, while the signing remains centralized. Unfortunately, public key signature verification has higher computational overhead than a symmetric one.

L. van Doorn, Ballintijn, and Arbaugh[14] described a practical implementation of asymmetric signed executables in Linux. To reduce the overhead associated with asymmetric signature verification, this work did extensive caching of the results of the verification. Thus the asymmetric signature on an executable is verified the first time, the result cached in memory, and reused all subsequent times, unless the file is modified, or the system rebooted, clearing the memory cache. This caching reduced the boot-time overhead of asymmetric signature verification from 75% to 5%, thus making its

use more practical. One limitation of this project was that the digital signatures were stored in the ELF headers, thus limiting signature verification to ELF executables only. Many security sensitive programs are implemented as shell or Perl scripts, which could not be verified.

Another category of executable checking is to look for signs of malicious code, such as with anti-virus or spyware checking programs. Anti-virus programs check executables, looking for signs of malicious code. Mechanisms for this checking include looking for known data patterns within the executable, and running the executable in an virtual machine during execution. All of these methods are very time consuming, so they are normally checked only periodically, not every time the executable is run. In addition, as new viruses are discovered, the table of known bad patterns must be updated, thus forcing rechecking of all executables. Thus malicious executables may be run, if they are introduced between scheduled checks.

Another security check is to determine that the executable is the most current. An executable may be signed, and checked for malware, but may still be a security threat because it is outdated, containing a known vulnerability fixed in a more recent version. Patch management systems have been developed to compare installed versions against a trusted list of current revisions, to ensure that all security critical patches have been applied. As in the anti-virus case, this is an expensive operation, requiring checking all executables against a remote list, so this is normally done only periodically, with the resultant security exposure between checks.

EVM provides an improved, extensible system with:

- a policy based verification function based on storage of verification data in authenticated extended attributes
- a single, symmetric key based verification function, with TPM protection of the key
- verification on every file open or exec
- integration with MAC based integrity containment, for flexible policy action
- the use of the TPM to bootstrap EVM integrity

EVM defines authenticated extended attributes to store a file's security meta data. The basic minimal EVM attributes include:

- security.evm.hash – a hash of the file's data
- security.evm.hmac – an HMAC of all security.evm (and security.slim) attributes

The EVM policy can define additional checks and associated security.evm attributes. The current EVM policy implements additional attributes to store additional RPM package information, including version and packager.

Since the verification header is stored in file system extended attributes, this verification approach can be used for any type of file, not just for executables. Files such as configuration files, or interpreted language scripts can be similarly verified and operations enforced. The only difference is that the policy is checked and enforced at file open time, instead of the normal check at file execution. In addition, the policy rules specify possibly different actions. Rather than rules that control execution, rules for files being opened can include actions that restrict modes or prohibit the open from succeeding. Since an executable file can be either opened or executed, both sets of rules could apply, depending on which operation was requested.

One disadvantage of using extended attributes for security meta data, is that extended attributes are not available for all file systems, particularly remote ones. IBM is working on a VFS module to layer extended attributes on top of arbitrary file systems, so this issue is not a long term problem.

In normal operation, when a new executable is installed, it is first checked by all of the verification methods listed in the EVM policy file, and the results inserted into the extended attribute list, along with a hash of the file, and HMAC of the attributes. At run time, the kernel then looks at the verification attributes and rapidly compares them to the current policy, and determines how to run it according to policy. Checking the header does require hashing the executable file, to verify that it hasn't been modified, but this hash, and subsequent symmetric key

HMAC is very fast compared to the original checking methods, and is cached until the next reboot. Thus the verification is done in optimal time, allowing checking on all accesses.

For the symmetric key HMAC, EVM needs a symmetric secret key. This key is provided by the TPM module, (derived from the kernel master key as described earlier.) This TPM based protection defends all of the authenticate extended attributes from off-line attack.

During implementation, we did run into one problem related to the use of “prelink”, a program which is used to modify executables so that they can be loaded faster. The problem is that prelink alters the executable file's hash, so that its integrity cannot be verified against the signature in the original rpm package. From a security perspective, periodic prelinking and thus alteration of a file's hash makes it too hard to verify integrity against the original package, and so we simply turned prelinking off. If prelinking is really desired, prelink would need to be modified to verify the hash of the base executable, before prelinking, and then to update the security.evm.hash attribute to the new value after prelinking.

EVM uses the Simple Linux Integrity Module (SLIM) mandatory access control module for access enforcement. Rather than simply refusing to run an executable which does not have all of the desired attributes, the system has the flexibility to run less trusted executables with correspondingly lesser MAC privileges. SLIM mechanisms provide just such an ability to constrain executable privileges, based on the executable's trust attributes.

Simple Linux Integrity Module (SLIM):

The EVM module verifies that all files are authentic, unmodified, current, and not known to be malicious. EVM does not (and cannot) determine if files are correct - that is that given any (possibly malicious) input data, that they will operate properly. A data driven compromise of the operation of verified files can still lead to the compromise of a system, despite EVM checking. An integrity enforcing model is needed to block, or at least contain any such compromise.

Integrity containment has been a well studied subject in mandatory access control. Biba [2] proposed an integrity mandatory access control model in 1977. In the traditional Biba model, processes and objects (files) are given integrity labels. Low integrity processes cannot write to high integrity objects, while high integrity processes cannot read or execute low integrity objects. In the low water-mark variation, a high integrity process is allowed to read or execute low integrity objects, but the process is automatically demoted to the object's level. Thus the process's integrity level is always the lowest level read/executed.

Tim Fraser implemented Lomac[3],[4], a low water-mark implementation for Linux. Lomac was exceptionally simple to administer, as it had only two levels (trusted and untrusted). All processes start out trusted until they attempt to read or execute an untrusted file or network socket, at which time they are demoted to low integrity. One problem with Lomac was that to be useful, it had to have some way to promote an untrusted object to being trusted, for example, to be able to install downloaded packages, or to allow logging in over ssh. Lomac allowed for object promotion by designating certain programs as "trusted". A trusted process could read untrusted objects without being demoted. Lomac designated these trusted programs simply by filename, and did not verify their integrity, offering potential attacks.

Caernarvon [6] is a modified Biba model which specifically incorporates support for verified trusted programs which are allowed to remain high integrity while reading low integrity objects. Caernarvon added some critical refinements: first, the abilities of a high integrity process to read and execute lower integrity objects are separated, so that trusted reading does not automatically grant trusted execution, and second, trusted programs are verified by digital signature.

Another option for mandatory access control integrity enforcement is the NSA's security enhanced Linux (selinux [7], [12]) module. This module allows the creation of a wide range of security models. Its default security model is Type Enforcement (TE), which is more powerful than Biba or Caernarvon, but is correspondingly more difficult to configure. A recent

paper [5] analyzed the default 33 thousand line selinux policy rule set, and discovered that despite its power and complexity, it did not ensure integrity containment. Also, selinux currently does not provide the functionality for low water- mark or high water-mark rules. A highly desirable future project could look at modifying selinux to be able to take advantage of EVM's trust findings for a file, and to be able to support SLIM's demotion and promotion rules. This would combine the best of EVM's file verification, and selinux's support for multiple MAC policies, and SLIM would become simply another loadable selinux policy. As selinux also uses extended attributes (security.selinux) for storing labels, EVM could authenticate those labels too, thus providing protection against integrity attacks on the labels.

SLIM builds upon a combination of Caernarvon and Lomac models, using low water-mark integrity handling like Lomac's, with Caernarvon's separation of trusted read/execute, and signed trusted programs. SLIM also uses the results of EVM's file verification to give trusted process authority only to those files which meet all the file verification requirements, including authenticity, integrity, currency, non-malicious content, and trusted program designation. Thus it significantly extends Caernarvon's trusted program signature verification. SLIM is very simple to administer, and provides greater ease of use, due to its support of low and high water-mark rules.

In SLIM, all files are labeled with the security.slim.level extended attribute to indicate:

Integrity Access Class (IAC): one of

SYSTEM
USER
UNTRUSTED
EXEMPT

Secrecy Access Class (SAC) : one of

SENSITIVE
USER
PUBLIC
EXEMPT

The EXEMPT classes are essential to handle objects such as /dev/null and /dev/zero, which all processes must

be able to access, and whose access is safe (absent a kernel vulnerability). This security.slim.level attribute is authenticated under security.evm.hmac, so it is protected from off-line attack.

In SLIM all processes inherit classes from their parents:

- Integrity Read Access Class (IRAC)
- Integrity Write/Execute Access Class (IWXAC)
- Secrecy Write Access Class (SWAC)
- Secrecy Read/Execute Access Class (SRXAC)

SLIM enforces the following Mandatory Access Control Rules:

Read:

- IRAC(process) <= IAC(object)
- SRXAC(process) >= SAC(object)

Write:

- IWXAC(process) >= IAC(object)
- SWAC(process) <= SAC(object)

Execute:

- IWXAC(process) <= IAC(object)
- SRXAC(process) >= SAC(object)

For most processes, IRAC == IWXAC and SWAC == SRXAC, which simplifies to the original Biba model.

SLIM extends this basic model with low water-mark integrity and high water-mark secrecy rules. On read, if a process with higher IRAC attempts to read an object of lower IAC, the process is demoted to that IAC, so that the read is allowed. Similarly, if a process of lower SRXAC attempts to read a higher secrecy object, the process is allowed to do so, but it is promoted to the higher SAC.

For trusted programs, the program file contains both the file's normal object labels (IAC, SAC), along with the special labels for the allowable range of IRAC, IWXAC, SRXAC, and SWAC for the process when executed. Thus trusted programs (suitably verified by EVM) can be run which are allowed to read lower integrity objects, and read higher secrecy objects without being correspondingly demoted or promoted.

An example of a trusted program is rpm. Normally, as rpm packages are received over the untrusted network,

they are initially labeled as untrusted. Any process (even with root user id) which attempted to read the rpm, would be demoted to untrusted, and would no longer be able to install the package. A small, simple guard program "promote" is used by rpm to verify the package's public key signature, and if it is verified, to relabel the file to the SYSTEM integrity class, so that the normal rpm process can install it. Promote is specially labeled so that it is allowed to read UNTRUSTED level files without demotion, as it is trusted to relabel them only if the signature verifies. Note that promote itself is first verified by EVM to have the valid trust characteristics (hash and hmac) before it is executed with the trusted process classes.

General Notes:

- Demotion/Promotion occur on attempts to read or execute, but not on attempts to write. Attempts to write to higher integrity or lower secrecy are always blocked.
- Demotion resets both integrity classes (IRAC and IWXAC).
- Promotion resets both secrecy classes (SRXAC and SWAC).
- If a process has appropriate execute classes (IWXAC and SRXAC) to execute a guard process, then all four of the process's access class values are replaced by the values specified in the guard executable labels.

Installation

In the long term, TLC should be integrated into the distribution's installation process, which would include adding the following steps:

- Initialization of the TPM chip
- Creation of the sealed kernel master key, with associated authorization password.
- verification and labeling of all files as each package is installed.

Currently, TLC is distributed in source form, and is installed on top of an existing system. This is done in two steps. First the sources are compiled and installed

into the respective module or application directories. Then a one-time installation step is done which initializes the TPM, creates the sealed kernel master key, adds all the EVM and SLIM labels, and creates a new init ramdisk. After the initial installation, all subsequent installation and labeling is done transparently under the normal RPM installation and update process.

Boot time

One goal of TLC has been to make booting and logging in as painless as possible, while improving the strength of authentication and integrity measurement. On our reference T42p, at boot time, the user is first presented with a BIOS prompt to swipe a finger on the fingerprint sensor. If successful, the fingerprint hardware processor provides BIOS with the system's power-on password, to enable booting, and hard-disk password, to enable access to the disk. Grub is then used to measure and load the kernel and init ramdisk images, and turns control over to the kernel. When the kernel runs the init script in the init ramdisk, it invokes the loadkernkey application, which prompts for username and password. The user's sealed kernel key is located on USB or in the init ramdisk, and it and the password are presented to the TPM driver. If the sealed key and password match, the TPM unseals the kernel master key, and an appropriate derivative is made available to EVM, so that EVM can verify all files on the hard disk, as they are accessed. If the unsealing is successful, the init script also creates a one-time file telling gdm (the graphical login program) that the user is already authenticated, directly starting the user's desktop. Thus the user has only to swipe one finger, and provide a user name and password, which are leveraged to cover all aspects of BIOS, hard disk, TPM, EVM, and user authentication.

Security Results

On-line attacks were attempted, through browser downloaded, and email attached malware. The malware, having come from the network was properly untrusted, and all attempts to execute it resulted in the process being demoted to untrusted. Subsequent attempts to alter user or system level files were blocked. This protection was true even for root user processes. Direct

attempts to alter system level files were blocked by SLIM, The more sophisticated attacks of trying to get a guard program like promote to relabel untrusted code failed, as they required finding a vulnerability in promote's code.

Analysis of TLC's effectiveness against off-line attacks was more interesting. While it is infeasible for an attacker to forge file labels in the existing system, the attacker could replace the entire hard disk with one containing a version which simply pretended to authenticate and load the kernel master key, and which then did no TLC checking. While this fake kernel could not obtain the real kernel master key (the TPM would refuse to unseal the key, as the PCR values would be different), it could be difficult for the user to notice any difference. To protect against this wholesale replacement attack, the user's home directory could be loopback encrypted with a key derived from the real kernel master key, so that the fake kernel could not present any of the user's environment or files to the user.

Performance Results

Worst case performance impact occurs at boot time and at first invocation of any large application, as hashing of file data is done (just once) at first open or execution. Subsequent accesses use the cached results, with negligible impact. We measured the total clock time for the entire boot sequence (subtracting any time waiting for the user), and for invocation of openoffice. For the boot sequence, times were measured with and without TLC. For the openoffice invocation, times were measured for first and subsequent invocations, with and without TLC. These worst case results were:

<i>Test</i>	<i>No TLC(sec)</i>	<i>TLC (sec)</i>	<i>delta(sec)</i>	<i>delta (%)</i>
Boot	50	55	5	10
Open Office first run	16	16	0	0
Open Office second	6	6	0	0

These results confirmed our expectation that the overhead of the hash and symmetric key HMAC were small, and largely confined to boot time.

Future Work

The most significant improvement would be to integrate the TPM/EVM extended verification with selinux, so that any selinux policy could benefit from both the authenticated extended attributes protection on the policy's labels, and from policies which take advantage of the results of extended verification on all files. In addition, if selinux were extended to support SLIM low and high water-mark rules, then SLIM could be implemented simply as another selinux policy.

Another useful extension would be to integrate TLC with one of the existing Linux anti-virus scanners, so that anti-virus scan results could be recorded and used in EVM and SLIM policies, with virtually no additional run-time performance penalty.

Summary

TLC protects the integrity of Linux client systems against a wide range of on-line and off-line attacks. It uses TPM hardware to establish initial integrity at boot time, to protect against off-line attacks, measures the integrity of all files with an efficient, extensible authenticated extended attribute mechanism, and uses this integrity information with a simple mandatory access control model to protect against on-line attacks. In testing, TLC has minimal overhead, and excellent transparency to the user.

Availability:

A TLC open source package will be made available at:
<http://www.research.ibm.com/gsal/tpca>

References:

[1] "AOL/NSCA Online Safety Study", October 2004,
<http://www.staysafeonline.info/news/NCSA-AOLIn-HomeStudyRelease.pdf>

- [2] K. J. Biba. "Integrity Considerations for Secure Computer Systems" Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, April 1977.
- [3] T. Fraser, "LOMAC: Low Water-Mark Integrity Protection for COTS Environments," Proceedings of the 2000 IEEE Symposium on Security and Privacy, Oakland, California, USA, 2000.
- [4] T. Fraser, "Lomac",
<http://opensource.nailabs.com/lomac>
- [5] T. Jaeger, R. Sailer, X. Zhang, "Resolving Constraint Conflicts" SACMAT '04 June 2-4 2004 Yorktown Heights, NY
- [6] P. Karger, V. Austel, and D. Toll. "Using a Mandatory Secrecy and Integrity Policy on Smart Cards and Mobile Devices" EUROSMART Security Conference. 13-15 June 2000, Marseilles, France p. 134-148.
- [7] NSA, "selinux", <http://www.nsa.gov/selinux>
- [8] M. Pozzo and T. Gray, "An Approach to Containing Computer Viruses", Computers and Security, 6(4) 321-331 August 1987.
- [9] D. Safford, J. Kravitz, L van Doorn, "libtpm"
<http://www.research.ibm.com/gsal/tpca>
- [10] D. Safford, J Kravitz, L van Doorn, "Take Control of TCPA", Linux Journal, August 2003.
- [11] SIMC Conference "Fishing and Other Impersonations", Jersey City, NJ , 2004
<http://www.simc-inc.org/archive0405/phishing/>
- [12] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. "The Flask Security Architecture: System Support for Diverse Security Policies" Proceedings of the 8th USENIX Security Symposium, pages 123139, Washington, DC, August 1999.
- [13] Trusted Computing Group
<http://www.trustedcomputinggroup.org>
- [14] L. van Doorn, G. Ballintijn, and W. A. Arbaugh, "Design and Implementation of Signed Executables for Linux",
<http://www.cs.umd.edu/~waa/pubs/cs4259.ps>