An Overview of The Linux Integrity Subsystem

IMA (Integrity Measurement Architecture) was introduced in Linux 2.6.30, as part of an overall Linux Integrity Subsystem, composed of a number of related components, including:

In-kernel:
- IMA
- IMA-Appraisal
- IMA-Appraisal-Directory-Extension
- IMA-Appraisal-Signature-Extension
- EVM
- Trusted and Encrypted Keys

Application/support:
- Trousers/Utilities
- OpenPTS

This document gives an overview of these components, their goals, features, design, status, and benefits. As these components are in varying states of completion, this document is intended to help see the final overall integrity architecture once all components are integrated.

Goals:

There are three main security goals for the programs and data which comprise a Linux system: Integrity, Authenticity, and Confidentiality. These goals are similar and tightly coupled, so it is helpful to define them clearly.

Integrity - state of being entire, complete, unbroken,
free from corrupting influence (Webster)

Integrity is the most fundamental of the security attributes; without integrity, a system cannot have authenticity or confidentiality, as the mechanisms for authenticity and confidentiality themselves could be compromised.

For a file, integrity is often thought of as the property of being "unchanged", that is, once installed, the file is not changed by malicious or undesired modification. Cryptographically, a hash can be used to detect if a file's content has been changed, and applications such as tripwire (http://sourceforge.net/projects/tripwire/) and Aide (http://sourceforge.net/projects/aide/) are often used by administrators to compare file hashes against a database of expected or "good" values, to look for unanticipated changes.

In the extreme case, integrity can mean not just unchanged, but "unchangeable", or "immutable", so that it is not possible for anything to modify the file. The BSD operating system has the concept of immutable files, which can only be changed in the administrative single user mode. Android keeps all of its system files in a partition which is mounted read-only during normal use, and which can only be modified by the recovery program during boot. The problem with these immutable systems is that while they provide greater integrity, they make it difficult to do normal installation or patching of system components.

IMA follows Trusted Computing Group (TCG) open integrity standards, which use a combination of these integrity techniques (hashing and immutability) to create a new integrity solution. In contrast to the BSD and Android approaches, which make the files themselves immutable, the TCG approach is to make the measured hashes of files immutable, by storing them in hardware such as the Trusted Platform Module (TPM), Mobile Trusted Module (MTM) or similar device. IMA maintains a list of hashes of all files as they are accessed, and if a supported hardware security component like a TPM is available, it's immutability to software attack is used to "attest" to, or anchor, the list of measurements with a hardware signature, for remote verification of the measurement list. In this way, an administrator can centrally monitor the integrity of networked systems, and can tell if they have been altered or compromised.

Authenticity - being of established authority for truth and correctness (Webster)

Authenticity is an extension of the concept of integrity. The idea here is to tell not only that a file has integrity, and is therefore unchanged, but also that the file is of known provenance. For example, we would know not only that a file has not been maliciously modified, but also that the file was the original file provided by a vendor, such as RedHat.

A simple hash does not suffice for authenticity. Instead a signature based on some secret is used, so that an attacker cannot forge the proof of provenance. Normally a public key signature such as RSA is used for authenticity, but symmetric keyed hashes, such as HMAC can also be used.

An advantage of public key signatures is that the central authority is the only entity who has to protect the confidentiality of the private key, and all other systems need only protect the integrity of the public key used for verification. The problem with public key signatures is that they are only practical for files which never change, since the local system cannot re-sign the files if they do change. Since many security critical files on a system change, these changing files can only be protected practically with a local symmetric key based signature. The local symmetric key used for authentication must be carefully protected, as an attacker could use it to forge a local signature on a maliciously changed file.

In Linux, a new IMA-Appraisal extension, and a new Extended Verification Module (EVM) are components which use a symmetric key to HMAC a file's data and metadata to provide authenticity for files which are allowed to change, and which therefore must be locally signed. For files which are not expected to change, an IMA-Appraisal-Signature extension will store an authority's RSA public key signature. Files with RSA signatures will not be allowed to change, other than to be deleted, and then only by a privileged process (a process which has been assigned the proposed new "integrity" capability.)

Since EVM uses a local symmetric key for authenticity, this key must be carefully protected. New "trusted" and "encrypted" kernel key types can be used to protect the EVM key, by sealing in a TPM, and releasing only under specified integrity measurement values.

Confidentiality - the state of being secret (Webster)

The final desired characteristic is confidentiality. It is a common misconception that encryption by itself can protect the integrity of files, and therefore all one needs is to encrypt the filesystem to achieve both integrity and confidentiality.

Unfortunately it's not that simple - encryption is not integrity. First, as a simple example consider a stream cipher such as RC4, which was for a long time used as the standard encryption for all wireless and internet (https) communication. RC4 by itself provides no integrity protection whatsoever, and it is trivial for an attacker to "bit-twiddle" any desired bits in the encrypted stream. Even with block ciphers, with the current standard chaining methods, encrypted blocks can be cut and pasted and replayed between files and sessions with a good probability of success. Second, there are significant performance penalties for encrypting system files, where integrity, not confidentiality is all that is required.

In IPSEC, a deliberate decision was made to allow integrity (AH - authentication headers) without encryption, but not encryption without integrity.

The bottom line is that encryption by itself does not guarantee integrity, and you really want both.

One of the hardest problems in confidentiality is key management. It does no good to have strong encryption with a key system that makes it easy for an attacker to steal the keys.

The trusted key component does two things to help with secure key management on Linux. First, it provides a kernel key ring service in which the symmetric encryption keys are never visible in plain text to userspace. The keys are created in the kernel, and sealed by a hardware device such as a TPM, with userspace seeing only the sealed blobs. Malicious or compromised applications cannot steal a trusted key, since only the kernel can see the unsealed blobs. Secondly, the trusted keys can tie key unsealing to the integrity measurements, so that keys cannot be stolen in an offline attack, such as by booting an unlocked Linux image from CD or USB. As the measurements will be different, the TPM chip will refuse to unseal the keys, even for the kernel.

In this way, Linux ties confidentiality and integrity together. The integrity of the system is a prerequisite for decrypting confidential files.

Threat Models:

There are many different types of attacks on file integrity, authenticity, and confidentiality, and there are significant tradeoffs between security and cost and performance, depending on which of the threats need to be defended against.

Remote Attacks

The most common attack is the remote software attack, in which the attacker attempts either to trick a user into running the attacker's malicious code (trojan), or in which the attacker sends malicious data in an attempt to exploit a vulnerability in the system's software (injection, overflow).

Local Attacks

Local attacks assume that the attacker has physical access to the system, such as with a malicious insider (operator), or in the case of theft of the system. Local attacks can be software based, such as in an offline attack, or can be hardware based, such as with simple JTAG memory probes,

or even extremely sophisticated (and expensive) attacks in which chips are taken apart to read out sensitive data.

The simplest and most common local attack is the offline attack in which an alternate operating system is booted, from CD or USB drive, and this operating system is used by the attacker to modify the target system. Offline attacks typically will simply attempt to crack existing passwords, or to insert known ones, so that the attacker can simply log in. In more sophisticated offline attacks, malicious code can be inserted, and left in place, to capture sensitive data such as banking passwords at a later time.

Specific goals:

While it would be ideal to prevent all of these attacks, it is critical at least to detect when such an attack has succeeded. The best way to protect the integrity of a Linux system is to use mandatory access control (MAC), such as SELinux or Smack to confine untrusted code and data. Even with MAC, some remote, and most local attacks (particularly offline) are still possible.

The Integrity subsystem goals are to detect any malicious changes to files, for all remote and local attacks, short of the extreme case of a local attacker with the access, time, and resources to attack the hardware, (particularly the TPM chip.) Malicious changes can include modification of data or metadata (e.g. renaming), replaying old files, and deleting files.

Implementation

IMA - Basic Measurement and Attestation

IMA has been included in the Linux kernel since 2.6.30.

IMA is an open source trusted computing component. IMA maintains a runtime measurement list and, if anchored in hardware (e.g. TPM), maintains an aggregate integrity value over this list. The benefit of anchoring the aggregate integrity value in the TPM is that the measurement list cannot be compromised by any software attack, without being detectable. Even if a malicious file is accessed, it's measurement is committed to the TPM before the file is accessed, and the malicious code cannot remove this measurement. If the malicious software compromises the attestation software, it cannot conceal it's presence, because it cannot forge a signature on a fake measurement list. Conversely, if there is no hardware anchor, the malicious code can easily create a fake list which cannot be detected. Hence, on a trusted boot system, IMA can be used to attest to the system's runtime integrity. Note that IMA measurement and attestation does not attempt to protect a system's integrity, its goal is at least to detect if such compromise has occurred, so that it can be repaired in a timely manner.

IMA measurements are enabled with the kernel command line parameter

```
ima_tcb=1
```

This starts a default policy which measures all normal files that are executed or mmap'ed execute, and all normal files which are read by a process with root uid.

A modified IMA policy can be loaded, and the policy can be based on LSM labels. For example, if running SELinux, a desirable rule is

```
dont_measure obj_type=var_log_t
```

so that log files are not measured.

The IMA measurement list can be read through an IMA securityfs file, normally mounted at: `/sys/kernel/security/ima/ascii_runtime_measurements,` and the measurement list entries look like:

```
PCR   template-hash                                 \
      filedata-hash                               filename-hint

10    7971593a7ad22a7cce5b234e4bc5d71b04696af4 ima \
      b5a166c10d153b7cc3e5b4f1eab1f71672b7c524 boot_aggregate
```

The hashes are extended into a PCR (by default PCR-10). The Trusted Computing Group has defined a standard integrity attestation XML format, PTS, which includes the measurement list and the TPM's signature on the value in PCR-10. A remote system can validate that the list results in the value in PCR-10, and that the TPM has signed this value. No matter how the system has been compromised, the malicious software cannot forge a valid measurement list and corresponding TPM signature, since the malicious software cannot get access to the TPM's private signature key, and any signature of PCR values in the TPM must include the measurement of any malicious file taken before the malicious file was accessed. The malicious code cannot "take back" its own measurement, and cannot forge a "clean" measurement signature.

IMA-Appraisal - Local file integrity checking

IMA-Appraisal has been posted several times to the LSM/LKML mailing lists, but has not been upstreamed, pending some additional extensions, including the signature and directory extensions described later.

IMA-Appraisal is a new extension to IMA in which IMA stores the "good" hash of an appraised file in the security.ima extended attribute, and verifies that the current measurement of the file matches this "good" value. If the values do not match, access is denied to the file. By default, security.ima contains a hash, not a keyed signature. This is convenient, particularly for changing files, but does not provide strong integrity and authenticity protection against offline and online attacks, as the hashes are easily forged.

IMA-Appraisal-Signature-Extension - Digitally signed files

The IMA-Appraisal-Signature-Extension for digitally signed files, is intended to augment the standard IMA-appraisal, with authority based digital signatures (RSA), to improve both the authenticity and immutability guarantees available for unchanging files. It has not yet been posted. (First post is expected in early 2011.)

In the IMA appraisal signature extension, the content of the security.ima extended attribute may be an RSA signature, normally as provided by the vendor. When this attribute is set, which

requires the integrity capability, the file is considered immutable, and the signature is verified before access. Signed files cannot be modified, but they can be deleted, and replaced, to allow updates. These signatures are difficult to forge by an attacker, since they require possession of the authority's private key. (The reference public key does need to be integrity protected against offline attack.)

IMA-Appraisal-Directory-Extension - Local directory Integrity

The IMA-Appraisal-Directory-Extension has not yet been posted. It is intended to augment file appraisal to include protections against file name change attacks and signed file replay attacks in which an old signed file is replayed in an offline attack.

The basic IMA-Appraisal uses either RSA signatures (for unchanging files) or hashes (for changing files) to verify the authenticity and integrity of a file's contents. There are still attacks in which valid file content can be misused by changing a file's metadata, which is not protected by the signature or hash. Examples of sensitive metadata include the filename, owner, and mode (access control) bits. For example, simply by renaming "rm" to "ls" an attacker can damage a system. File metadata exists both in the directory (the file's name), and in the inode (owner, group, mode, LSM labels). To protect the file's metadata, the directory extension adds a hash of the directory's contents (filenames, inode numbers, and inode appraisal value).

EVM - Extended Verification Module - Trusted Metadata

EVM has been posted to the LSM/LKML mailing lists. Its goal is to protect sensitive inode metadata against offline attack.

The IMA-Appraisal-Directory-Extension protected filename metadata. The second area with sensitive metadata is in the inode, which contains owner, group, mode, along with sensitive extended attributes. EVM Protects these security extended attributes against offline attacks. EVM maintains a HMAC-sha1 across a set of security extended attributes, storing the HMAC as an extended attribute 'security.evm'. To verify the integrity of an extended attribute, EVM exports evm_verifyxattr(). EVM provides protection for all security extended attributes, including

* security.ima (IMA's hash or signature for the file)
* security.selinux (the selinux label/context on the file)
* security.SMACK64 (Smack's label on the file)
* security.capability (Capability's label on executables)

Since the EVM key used to HMAC the metadata is very sensitive, it should be a trusted key, or an encrypted key under a trusted key, and the trusted key should be integrity locked to the current measurement values.

Trusted and Encrypted Keys:

The Trusted and Encrypted key patches are currently in security #next, queued for potential inclusion in 2.6.38, to provide trusted keys for sensitive uses, such as for the EVM master key. Trusted and Encrypted Keys are two new key types added to the existing kernel key ring service. Both of these new types are variable length symmetric keys, and in both cases all keys are

created in the kernel, and user space sees, stores, and loads only encrypted blobs. Trusted Keys require the availability of a Trusted Platform Module (TPM) chip for greater security, while Encrypted Keys can be used on any system. All user level blobs, are displayed and loaded in hex ascii for convenience, and are integrity verified.

Trusted Keys use a TPM both to generate and to seal the keys. Keys are sealed under a 2048 bit RSA key in the TPM, and optionally sealed to specified PCR (integrity measurement) values, and only unsealed by the TPM, if PCRs and blob integrity verifications match. A loaded Trusted Key can be updated with new (future) PCR values, so keys are easily migrated to new pcr values, such as when the kernel and initramfs are updated. The same key can have many saved blobs under different PCR values, so multiple boots are easily supported.

By default, trusted keys are sealed under the SRK, which has the default authorization value (20 zeros). This can be set at takeownership time with the trouser's utility:

```
tpm_takeownership -u -z
```

Trusted Key Usage:

```
keyctl add trusted name "new keylen [options]" ring
keyctl add trusted name "load hex_blob [pcrlock=pcrnum]" ring
keyctl update key "update [options]"
keyctl print keyid

options:
   keyhandle=  ascii hex value of sealing key default 0x40000000 (SRK)
   keyauth=    ascii hex auth for sealing key default 0x00...
               (40 ascii zeros)
   blobauth=   ascii hex auth for sealed data default 0x00...
               (40 ascii zeros)
   blobauth=   ascii hex auth for sealed data default 0x00...
               (40 ascii zeros)
   pcrinfo=    ascii hex of PCR_INFO or PCR_INFO_LONG (no default)
   pcrlock=    pcr number to be extended to "lock" blob
   migratable= 0|1 indicating permission to reseal to new PCR values,
               default 1 (resealing allowed)
```

"keyctl print" returns an ascii hex copy of the sealed key, which is in standard TPM_STORED_DATA format. The key length for new keys are always in bytes. Trusted Keys can be 32 - 128 bytes (256 - 1024 bits), the upper limit is to fit within the 2048 bit SRK (RSA) keylength, with all necessary structure/padding.

Encrypted keys do not depend on a TPM, and are faster, as they use AES for encryption or decryption. New keys are created from kernel generated random numbers, and are encrypted or decrypted using a specified 'master' key. The 'master' key can either be a trusted-key or user-key type. The main disadvantage of encrypted keys is that if they are not rooted in a trusted key, they are only as secure as the user key encrypting them. The master user key should therefore be loaded in as secure a way as possible, preferably early in boot.

Encrypted Key Usage:

```
keyctl add encrypted name "new key-type:master-key-name keylen" ring
keyctl add encrypted name "load hex_blob" ring
keyctl update keyid "update key-type:master-key-name"
```

where 'key-type' is either 'trusted' or 'user'.

Examples of trusted and encrypted key usage:

Create and save a trusted key named "kmk" of length 32 bytes:

```
$ keyctl add trusted kmk "new 32" @u
440502848

$ keyctl show
Session Keyring
       -3 --alswrv    500   500  keyring: _ses
 97833714 --alswrv    500    -1   \_ keyring: _uid.500
440502848 --alswrv    500   500       \_ trusted: kmk

$ keyctl print 440502848
0101000000000000000001005d01b7e3f4a6be5709930f3b70a743cbb42e0cc95e18e915
3f60da455bbf1144ad12e4f92b452f966929f6105fd29ca28e4d4d5a031d068478bacb0b
27351119f822911b0a11ba3d3498ba6a32e50dac7f32894dd890eb9ad578e4e292c83722
a52e56a097e6a68b3f56f7a52ece0cdccba1eb62cad7d817f6dc58898b3ac15f36026fec
d568bd4a706cb60bb37be6d8f1240661199d640b66fb0fe3b079f97f450b9ef9c22c6d5d
dd379f0facd1cd020281dfa3c70ba21a3fa6fc2471dc6d13ecf8298b946f65345faa5ef0
f1f8fff03ad0acb083725535636addb08d73dedb9832da198081e5deae84bfaf0409c22b
e4a8aea2b607ec96931e6f4d4fe563ba

$ keyctl pipe 440502848 > kmk.blob
```

Load a trusted key from the saved blob:

```
$ keyctl add trusted kmk "load `cat kmk.blob`" @u
268728824

$ keyctl print 268728824
0101000000000000000001005d01b7e3f4a6be5709930f3b70a743cbb42e0cc95e18e915
3f60da455bbf1144ad12e4f92b452f966929f6105fd29ca28e4d4d5a031d068478bacb0b
27351119f822911b0a11ba3d3498ba6a32e50dac7f32894dd890eb9ad578e4e292c83722
a52e56a097e6a68b3f56f7a52ece0cdccba1eb62cad7d817f6dc58898b3ac15f36026fec
d568bd4a706cb60bb37be6d8f1240661199d640b66fb0fe3b079f97f450b9ef9c22c6d5d
dd379f0facd1cd020281dfa3c70ba21a3fa6fc2471dc6d13ecf8298b946f65345faa5ef0
f1f8fff03ad0acb083725535636addb08d73dedb9832da198081e5deae84bfaf0409c22b
e4a8aea2b607ec96931e6f4d4fe563ba
```

Reseal a trusted key under new PCR values:

```
$ keyctl update 268728824 "update pcrinfo=`cat pcr.blob`"
$ keyctl print 268728824
010100000000002c0002800093c35a09b70fff26e7a98ae786c641e678ec6ffb6b46d805
77c8a6377aed9d3219c6dfec4b23ffe3000001005d37d472ac8a44023fbb3d18583a4f73
d3a076c0858f6f1dcaa39ea0f119911ff03f5406df4f7f27f41da8d7194f45c9f4e00f2e
df449f266253aa3f52e55c53de147773e00f0f9aca86c64d94c95382265968c354c5eab4
9638c5ae99c89de1e0997242edfb0b501744e11ff9762dfd951cffd93227cc513384e7e6
```

```
e782c29435c7ec2edafaa2f4c1fe6e7a781b59549ff5296371b42133777dcc5b8b971610
94bc67ede19e43ddb9dc2baacad374a36feaf0314d700af0a65c164b7082401740e489c9
7ef6a24defe4846104209bf0c3eced7fa1a672ed5b125fc9d8cd88b476a658a4434644ef
df8ae9a178e9f83ba9f08d10fa47e4226b98b0702f06b3b8
```

Create and save an encrypted key "evm" using the above trusted key "kmk":

```
$ keyctl add encrypted evm "new trusted:kmk 32" @u
159771175

$ keyctl print 159771175
trusted:kmk 32 2375725ad57798846a9bbd240de8906f006e66c03af53b1b382dbbc55
be2a44616e4959430436dc4f2a7a9659aa60bb4652aeb2120f149ed197c564e024717c64
5972dcb82ab2dde83376d82b2e3c09ffc

$ keyctl pipe 159771175 > evm.blob
```

Load an encrypted key "evm" from saved blob:

```
$ keyctl add encrypted evm "load `cat evm.blob`" @u
831684262

$ keyctl print 831684262
trusted:kmk 32 2375725ad57798846a9bbd240de8906f006e66c03af53b1b382dbbc55
be2a44616e4959430436dc4f2a7a9659aa60bb4652aeb2120f149ed197c564e024717c64
5972dcb82ab2dde83376d82b2e3c09ffc
```

The initial consumer of trusted keys is EVM, which at boot time needs a high quality symmetric key for HMAC protection of file metadata. The use of a trusted key provides strong guarantees that the EVM key has not been compromised by a user level problem, and when sealed to specific boot PCR values, protects against boot and offline attacks. Other uses for trusted and encrypted keys, such as for disk and file encryption are anticipated. (A patch for ecryptfs is in process.)

Application Level Utilities

Trousers and Utilities

While the kernel includes a basic TPM device driver, and the kernel includes trusted keys, which directly use the TPM for key management, there are libraries and utilities needed for access and configuration of the TPM. Trousers provides a standards compliant TPM access library and associated TPM utilities for initialization, management, and use of the TPM.

OpenPTS

OpenPTS uses the Trousers library to access the TPM and the IMA measurement list and to create reference and current integrity manifests, with signed TPM quotes to anchor (authenticate) the measurement list. These reports are created in the PTS standardized formats for interoperability between applications and vendors.

Use Cases:

So what are the benefits of the Integrity Subsystem, and what are the scenarios where these benefits are obtained?

Host Integrity - providing assurance that managed hosts have not been compromised by remote or insider software attack.

> In the past, periodic measurement of file integrity (such as with Tripwire and Aide) has suffered from two problems. First it has performance problems with periodic hashing of all files, and second, as application level software, it is highly susceptible to compromise itself. With IMA, measurement is efficiently maintained in the kernel, cached along with the inode data. Integrity management applications, such as Aide can obtain all of the cached measurements, so that they do not need to perform redundant reading and hashing of already accessed files. Aide can also use OpenPTS to generate reference and current integrity reports which are formatted to TCG standards, and which are cryptographically verifiable based on the TPM's hardware signature, so that even if the system is compromised, the measurement list will be verifiable.

Host Authenticity - providing assurance that all files are authentic.

> For some use cases, such as embedded mobile devices, authenticity, in addition to integrity is an important requirement. Previous authenticity mechanisms, such as signed executables, (DigSig) have provided only a partial solution, working only on ELF executable files. With IMA appraisal and EVM, all files including scripts and configuration files can be authenticated, and protected against offline attacks. For unchanging files, an authoritative digital signature provides authenticity and immutability. For locally generated/changed files, the EVM symmetric signature protects local authenticity against offline attack.

Trusted encryption - providing more secure encryption by tying encryption keys to system integrity, so that sensitive data will not be exposed to compromised systems.

> Trusted keys bring CCA-like key management and protection to Linux, so that keys are never visible to userspace and applications, and so that the use of keys are tied to the system's integrity. Unlike previous trusted computing based keys, these keys are easily managed with the standard kernel key ring utilities, and are easily migrated across system updates and multiboot scenarios.

Trusted Network Connect (TNC) – PTS-based assurance of end to end trusted state during setup of a network connection between trusted nodes.

> For enterprise level customers, TNC based network admission control is essential to block compromised or misconfigured machines from accessing a sensitive network. By basing network admission on the IMA/PTS based integrity attestation, enterprises can have a strong hardware based integrity verification based on open TCG standards.

Summary:

This document has provided an overview of the integrity related kernel and application level components, their goals, design, and uses, to help understand the overall protections and benefits they provide.