Validated Execution Design

Scott Rotondo
Sun Microsystems, Inc.

1 Overview

Validated Execution provides a way to verify the integrity of program and library objects at the time of execution. Such verification provides assurance that the executable has not been altered, either accidentally or deliberately, since it was released by its publisher (which may be Sun, a third-party ISV, or the end customer).

There have been other attempts to solve this problem, such as Tripwire or `bart`, by comparing files against a manifest of known valid files. However, there is no protection against programs that are modified after the comparison but before they are executed. Performing the verification at the time of use avoids the race condition inherent in solutions that periodically compare files against a known reference.

Each executable file object is protected by a digital signature, which is constructed from a one-way hash of the file contents encrypted with a private RSA key known only to the publisher of the software. The corresponding public key can be used to decrypt the hash value and verify that it matches the file contents. Most ELF-format files in Solaris already contain an embedded digital signature; for those that do not and for non-ELF objects such as shell scripts, we provide a separate manifest listing the signatures for each file. This mechanism extends easily to allow third-party software vendors and end customers to create their own manifests of file signatures.

The general strategy employed is for entities that cause code to be loaded and executed, such as the kernel and run-time loader, to verify such code before execution. This establishes a chain of trust where each code component is validated by a previously validated component. To validate the earliest executing code in the system and initialize this process, we rely on a hardware component called a Trusted Platform Module (TPM) [1]. For systems without a TPM, all of the functionality described herein is still available, but the user must implicitly trust that the earliest code has not been modified.

This feature provides some degree of protection even against an attacker who is able to run processes with all privileges. We cannot protect in general against such a process, because it is able to alter any part of the file system or memory. However, the privileged process cannot alter the information stored in the TPM without knowledge of the TPM's owner password. Even if the running system is compromised, any alteration to the file system will be detected the next time the system is booted. Any executable files modified by the attacker will fail to start when the system reboots. Therefore, if an attacker gains privilege by exploiting a vulnerability, he must do so every time the system is booted rather than leaving behind a “back door” that can be used in the future to gain privileged access to the system.

2 File Validation

Validation of executables occurs in response to operations such as exec(), mmap() with `PROT_EXEC`, dlopen(), and loading a kernel module. Figure 1 contains a block diagram to illustrate the file validation process.

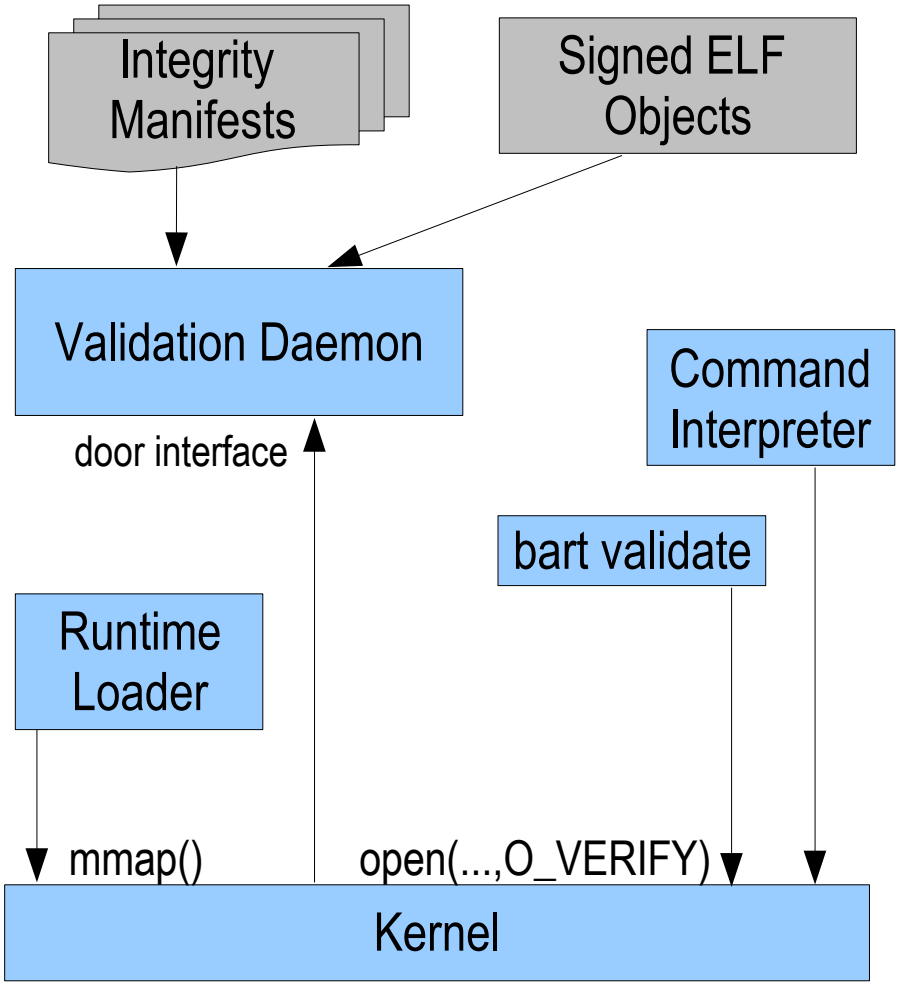

Figure 1. File validation block diagram

The kernel opens the executable file and then uses a door interface to contact the `signedexecd` daemon to perform the validation. The daemon validates the requested file either using its embedded ELF signature or by looking up the file's hash in a table that is initialized by reading the signature manifests installed on the system. Validation failures cause the new error code `EVALIDATE` to be returned.[1]

Although the mprotect() system call performs no direct validation, it is affected by the the maximum permissions set by an earlier mmap() call. The intent is to allow mprotect() to add `PROT_EXEC` only in situations where mmap() with `PROT_EXEC` would have been allowed. To that end, mmap() and mprotect() take the following actions:

1. mmap() without `PROT_EXEC` sets the maximum permissions to disallow `PROT_EXEC` if the unvalidated privilege cap is set to `noexec` (in addition to the existing reasons).

1 The error code `EACCES` and `ENOEXEC` were also attractive choices. However, `EACCES` can also mean that the execute permission bit is missing, and at least some shells will read and interpret a shell script directly if exec() fails with `ENOEXEC`.

2. mprotect() to add `PROT_EXEC` follows the rules for unvalidated objects described in section 5.3. It fails if the process's Permitted set exceeds the unvalidated privilege cap. Otherwise, the process's Limit set is intersected with the privilege cap.

2.1 File Manifests

Manifests containing file hashes are delivered with each Solaris release and patch to cover the files contained therein. Manifests are stored on the running system in the directory hierarchy rooted at `/etc/signedexec/manifest`.

The directory `/etc/signedexec/revocation` contains manifests that prevent the validation of any files that match their manifest entries. This provides the administrator with a mechanism to prevent the execution of programs that are later found to contain security vulnerabilities.

Each manifest is an XML file that conforms to the TCG specification for a reference manifest [2]. For each file, the manifest lists the following:

- full pathname (for identification only; not used for validation)
- basic file metadata (e.g. Owner, group, size, permissions)
- cryptographic hash of the entire file contents
- for ELF files, cryptographic hash of loadable sections only, corresponding to the hash used in the embedded ELF signature

The TCG-specified format provides for individual sections of the XML manifest to be signed separately, although Sun-generated manifests will include all entries in a single signed section. Because the signature covers only a portion of the file contents, it is possible to concatenate multiple manifests without invalidating their signatures.

2.2 Validation Algorithm

When the kernel needs to validate a file, it calls the internal routine signedexec_validate(), which sends a request to the validation daemon using a door interface. If the daemon is not available to respond to the request, as is the case early in the boot process before the daemon is started, signedexec_validate() falls back on the boot validation mechanism described in the next section.

A normal validation request proceeds as follows:

1. If the file has an embedded ELF signature:
   a. Find the public key certificate referenced in the embedded ELF signature section. If the certificate is not available, skip to step 2.
   b. Calculate the cryptographic hash of the loadable ELF sections only.
   c. Look up the hash in the table of revocation manifest entries. If a match is found, the validation fails.
   d. Verify the embedded signature of the hash. The validation succeeds if and only if the signature verifies successfully.
2. For all other files:
   a. Calculate the cryptographic hash of the entire file contents.

b. Look up the hash in the table of revocation manifest entries. If a match is found, the validation fails.

c. Look up the hash in the table of standard (non-revocation) manifest entries. If a match is found, the validation succeeds.

In the special case of a setuid or setgid exec() operation that changes the execution context of the process[2], the validation request includes a flag to require verification of the file's owner, group, and mode as well as contents hash. In this case, the validation is performed using the manifest table lookup (step 2 above) even if the file contains an embedded ELF signature. The file's owner, group, and mode must match those in the manifest entry, if present, or the validation fails.

The algorithm above results in one of three possible outcomes for a given file object.

- **Valid**: Loading and execution of the file object proceeds as requested.
- **Invalid**: The file explicitly fails validation, either because of a revocation entry or because there was a mismatch of file ownership or permissions during a setid execution.
- **Not validated**: There is insufficient information for the validation to explicitly succeed or fail. Code that is executed from such a file is subject to the semantics of the unvalidated privilege cap discussed in section 5.3.

2.3 Caching of Validation Results

When a file is successfully validated, this fact is cached on the vnode representing the file so that subsequent uses of the file do not incur the cost of validation. When the file contents or metadata are modified, as indicated by a write operation to the vnode, the cached results are invalidated.

However, the operating system is only able to directly intercept content changes that occur as a result of write operations on the local system. A file that is validated on an NFS client may be modified on the server without the knowledge of the client system. Subsequent reads from the server will return the modified contents. Similarly, disk devices that appear local to the operating system may actually be attached via a Storage Area Network (SAN) which provides alternative paths by which the disk contents may be modified.[3]

If a file system is subject to modification without a local write operation, we refer to that as a *volatile* file system, regardless of whether it is an example of the NFS or SAN case described above. If a validated file is modified, the operating system must detect or prevent use of the modified contents. Two possible ways to accomplish this are by recording a hash of each page of the remote file in order to detect changes or by locking all pages of the remote file in memory so that subsequent reads will be satisfied from pages cached in memory. A hybrid approach is also possible: Validated volatile files can be locked in memory until the system is under severe memory pressure and then replaced by page hashes. The initial implementation will implement only locking pages in memory. After experience with this approach, we may implement the page hash technique as well.

File systems that require this detection of content changes are identified via the `volatile` mount option. This option is specified by default for NFS mounts, though the administrator can override it by

2 The execution context changes only if the process's user id, group id, or Permitted privilege set changes as a result of the exec().

3 Another alternative path for modifying file contents exists even for locally-attached disks. A sufficiently privileged process could open the disk device containing a mounted file system and alter the contents of files that have already been validated.

specifying the `novolatile` option. For other file systems that may be remote, such as the SAN scenario, the administrator can choose whether to mount using the `volatile` option or not. In the case of a SAN that is physically secure, for example, the administrator might choose to accept the risk of file modification and avoid the performance penalty by mounting the file system without the `volatile` option.

As described in a later section, there may be different trust anchors (i.e. different certificates trusted to sign manifests) in different zones, and therefore the validation results may be different for different zones. Therefore, where a single file is mounted in multiple zones using lofs mounts, the validation results must be cached separately on the lofs vnodes and the underlying real vnode.

Each validation result includes a generation number. For the real vnode, the generation number is incremented whenever the file is written. For lofs mounts, the cached generation number matches the generation number of the underlying real vnode when the validation was performed. Before using the cached validation result from a lofs vnode, the system must verify that the generation number in the lofs vnode still matches the underlying vnode.

2.4 System Call Interface

A new signedexec() system call provides the following operations for use by the validation daemon. Each of these operations requires the `sys_admin` privilege.[4]

- **Enable or disable validation**: This is a zone-wide setting to enable or disable automatic validation of files upon execution.
- **Flush validation cache**: The kernel removes cached validation results from all vnodes.
- **Set privilege cap**: Set the maximum privilege set used for unvalidated executables. See the Administration section for details about how this privilege set is used.
- **Register door interface**: Provide a file descriptor for the door used to communicate with the validation daemon.

The signedexec() system call also includes the following unprivileged operations:

- **Get privilege cap**: Return the current value of the unvalidated privilege cap.
- **Validate a file**: Attempt to validate a file object as if it were being loaded for execution and return the result.

An SMF refresh operation will cause the validation daemon to reload its manifests. This is useful when manifests are added or removed from the system. The refresh operation also results in flushing the validation cache and setting the unvalidated privilege cap.

3 Bootstrapping and Chain of Trust

A significant number of user and kernel executables must be loaded before the validation daemon is running and available to respond to validation requests. Figure 2 illustrates the chain of trust that is used to validate objects from initial power-on until the system reaches its steady state to support validation as described in the previous section.

4 The `sys_config` privilege could have been appropriate instead, but it is not available in non-global zones.

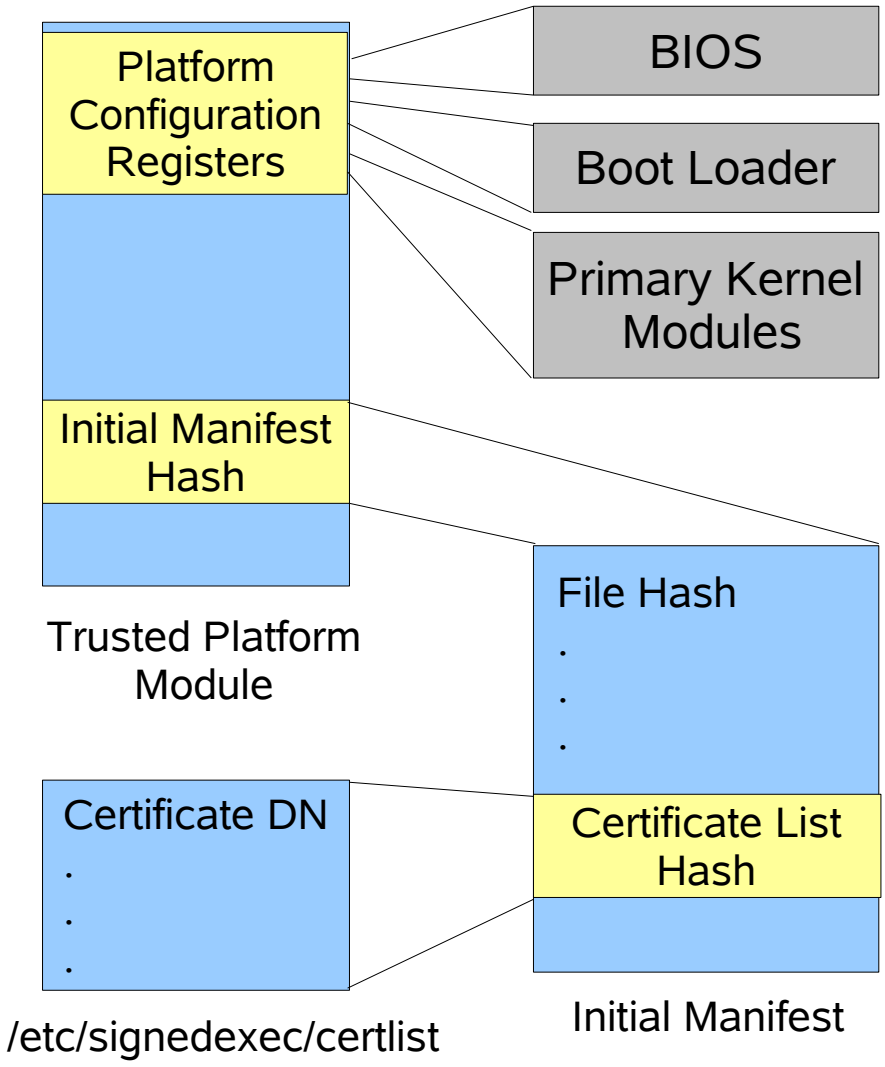

Figure 2. Chain of trust

Before the validation daemon is available, files are validated by the kernel directly using a simple *initial manifest*. This manifest contains only hashes of file contents, along with the file's ownership and permissions to allow for validation of setuid and setgid executions as described in the previous section.[5]

The manifest includes hashes for all files that may need to be executed before `signedexecd` is running and fully initialized. If the validation daemon later becomes unavailable for any reason, validation requests automatically fall back to using the initial manifest. Therefore, the initial manifest

5 There is a temptation to simplify the initial manifest structure by omitting the ownership and permission metadata and allowing setuid executions to fail when the initial manifest is used for validation. However, the initial manifest is used when the validation daemon unexpectedly fails, and one of the first things a user is likely to do to repair such a failure is to run /bin/su, which is a setuid program.

also needs to contain entries for utilities and libraries needed to repair the system.

One of the entries in the initial manifest is used to validate the file `/etc/signedexec/certlist`, which contains a list of the certificates (by distinguished name) that are valid for signing manifests used by the validation daemon. The problem of protecting the filesystem contents therefore reduces to protecting the initial manifest. All executable objects are either validated directly using the initial manifest, or they are validated using a manifest that is signed by a certificate enumerated in a file protected by the initial manifest.

Construction of the initial manifest is similar to the boot archive used to boot the system. Like the boot archive, it is generated from all of the files found in a small number of directories, including `/kernel`, `/platform`, `/lib`, and `/sbin`. The administrator must generate and sign a new initial manifest when files in these directories are modified, for example after applying patches to the kernel.

The initial manifest itself is included in the boot archive since it is needed for validating the executable files that are loaded from the boot archive. As described in section 5, the state of the SMF `signedexec` service is used to globally enable or disable file validation. Since SMF state information is not available when the kernel begins executing, the boot archive contains an additional file called `/etc/signedexec/disable` if validation is to be disabled.

3.1 Trusted Platform Module

On systems containing a Trusted Platform Module, the TPM is used to protect the initial manifest and to validate the firmware and boot-loader code that executes before the operating system. The TPM records measurements of all code that executes from initial power-on in its Platform Configuration Registers (PCR's). Beginning with an immutable portion of the system firmware, each piece of code in the boot sequence computes the hash of the code it is about to launch and stores that measurement in the appropriate PCR. The table below shows the PCR assignments that have been defined by TCG[3].

PCR	**Use**
0-1	Firmware code & data
2-3	Option ROM code & data
4-5	Boot loader code & data

The TPM also includes NVRAM storage locations that can be protected in two ways. Reading or writing can be configured to require knowledge of the TPM owner password, or particular PCR contents, or both.

The initial manifest hash is stored in a protected NVRAM location, whose address is included in the manifest. The NVRAM location is configured so that it can be modified only with knowledge of the TPM owner password, and it is also “sealed” using the PCR values that correspond to the firmware, boot loader, and any kernel modules that run before the initial manifest hash is verified. As a result, verifying the hash of the initial manifest also implicitly verifies the validity of all code that executed before that point.[6]

6 For the “failsafe” boot archive, however, it is advisable to avoid including the PCR's that represent pre-OS code when sealing the manifest hash. This ensures that the failsafe archive is completely self-contained so that it will boot correctly even if the firmware or boot loader code is modified. Alternatively, the failsafe archive could simply disable automatic file validation.

3.2 Limitations

There is, of course, a need to implicitly trust the installation media when the system is first installed. Taking ownership of the TPM to establish the TPM owner password requires the use of operating system software, either during installation or at some subsequent time. In either case, the operating system running at that time must be trustworthy, or the user's communication with the TPM may be subverted.

Even with a Trusted Platform Module, there is nothing to prevent booting a system with an operating system that does no validation, such as a previous release of Solaris.[7] The TPM will accurately record the pre-OS software measurements in its PCR's, but it has no active role in determining which code can execute. However, access to user data can be denied if a non-validating OS is booted by using a technique like the BitLocker feature in Windows Vista [4], where the disk encryption key is stored in the TPM and released only if the PCR's have the expected contents. This project allows for similar storage of a Solaris filesystem encryption key but does not require it.

4 Data File Validation

Previous sections have focused on the automatic validation of executable files. However, there are cases where programs may wish to validate other files, such as data files used to store configuration information or scripts that are interpreted by the program. A program triggers this validation by specifying the `O_VERIFY` flag when opening a file. The privilege sets of the process and the unvalidated privilege cap described below have no effect on the file validation described in this section.

When a file is opened using `O_VERIFY`, the kernel validates the file as if it were an executable using the same door interface described in section 2. The open operation succeeds only if the validation attempt is successful; otherwise the `ENOEXEC` error code is returned. Any attempt to use `O_VERIFY` on an object other than a regular file returns an `EINVAL` error code. An indication of the successful validation is cached on the vnode representing the file, as it would be for an executable.

The file table entry is also marked to indicate that it was opened using `O_VERIFY`. Reading from such a file descriptor causes the file to be validated again, but this validation normally succeeds immediately due to the cached results stored on the vnode. If the file has been written since it was validated, the cache is invalidated, and the file is validated again. The read operation fails with `ENOEXEC` if the validation is not successful.

This approach allows a program to rely on the contents of a file opened with `O_VERIFY` without having to explicitly check for subsequent modifications after the file is opened. It also allows data files to be updated in a controlled manner, where the file is modified and a manifest reflecting the new file contents is provided to the validation daemon before any consumer of the file attempts to read it again.

5 Administration

5.1 SMF Configuration

Most administration of the Validated Execution subsystem is accomplished by manipulating the state

7 This risk is partially mitigated by BIOS or bootprom options like the `security-mode` setting in Sun SPARC systems that restrict the choice of operating systems that can be booted. However, they are ineffective if the disk drive can be removed and accessed on another system.

and property values associated with the SMF `signedexec` service. This service exists in both global and non-global zones; enabling or disabling the service determines whether automatic validation of files is active in the zone.

When the enabled state of the `signedexec` service changes, the SMF start or stop method notifies the kernel using the signedexec() system call. In the global zone, it also adds or deletes the file `/etc/signedexec/disable` in the boot archive so that the kernel can determine whether to validate files early in the boot process.

A validation daemon runs in the global zone whenever the `signedexec` service is enabled. In a non-global zone, a separate instance of the validation daemon is started if the service is enabled and the `use_global_settings` service property is set to `false`. When this property is set to `true`, validation requests within the non-global zone are handled by the daemon running in the global zone, and other property settings are ignored. See section 5.4 for more details about the operation of non-global zones.

The `unvalidated_privilege_cap` property determines the maximum privilege set available to an executable that fails validation, as described in section 5.3 below. This property is ignored in a non-global zone if `use_global_settings` is set to `true`.

5.2 Certificate Storage

Certificates used to verify manifest signatures are stored in the `/etc/certs` directory. This collection of certificates determines the set of entities that are trusted by the system administrator to sign manifests and executables. When the validation daemon runs in a non-global zone, it uses certificates stored in the `/etc/certs` directory within the zone's root file system.

5.3 Maximum Privilege Set for Unvalidated Objects

The value of the unvalidated privilege cap takes one of two forms. It may specify a privilege set following the normal syntax for privilege sets, or it may be set to the distinguished value `noexec` which indicates that the unvalidated executable may not run at all.

When an attempt is made to load an unvalidated object for execution, the effect is as follows:

- The operation fails if the unvalidated privilege cap is set to `noexec`.
- The operation fails if the process's Permitted set exceeds the unvalidated privilege cap.
- If neither of the above conditions applies, the process's Limit set is intersected with the unvalidated privilege cap.

Informally, this means that the privilege cap constrains the maximum privileges available to this program or any of its descendents, as indicated by the process's Limit set, but the Limit set will not be reduced below the Permitted set. Instead, the operation will fail if the Permitted set exceeds the cap.

The kernel implicitly runs with all privileges. Therefore, an attempt to load an unvalidated kernel module fails unless the unvalidated privilege cap is equal to `all`.

5.4 Non-Global Zone Administration

There are two different paradigms for zone usage in Solaris. In the first case, individual zones are used

as a replacement for separate systems. Each zone is administered separately, and the primary purpose of the global zone is to allocate resources and otherwise manage the individual zones. A second model, typified by Trusted Extensions, contains multiple cooperating zones that are administered as a single system. To accommodate these two administrative models, we provide two options for the running of the validation daemon.

One instance of the validation daemon always runs in the global zone. Its unvalidated privilege cap and its set of signed manifests determine the success or failure of validations for the global zone and any other zone that does not run its own instance of the validation daemon. Validation requests in a non-global zone use the validation daemon running in that zone if present and otherwise fall back to the global zone daemon.

For administrative convenience, this allows non-global zones to inherit the privilege cap and manifests of the global zone wherever that is desired by simply setting the `use_global_settings` property in the non-global zone. If one or more zones require different settings, they can run local copies of the validation daemon. A non-global zone could also inherit the manifests of the global zone but specify its own privilege cap by running a local validation daemon but using a loopback mount for `/etc/signedexec`.

6 BART

The existing `bart(1)` utility is extended to provide the administrative interface for operations that deal with file manifests. The existing utility consists of two subcommands, `bart create` for generating a manifest from the current contents of the file system, and `bart compare` for comparing two previously-generated manifests and reporting the differences between them.

6.1 Manifest Generation

The default format for manifests generated by `bart` is modified to follow the XML format described in section 2. This allows software developers and end customers to use `bart` to generate manifests that can be consumed by `signedexecd`. Two new options are added to the `bart create` subcommand:

`-a` *alg* Use algorithm *alg* for file content hashes. Uses the same hash algorithm names as `digest(1)`.

`-p` Generate a plain-text manifest suitable for use with previous versions of `bart`. Implies `-a md5`.

For compare operations, old-format manifests are interchangeable with new-format manifests generated with the MD5 hash algorithm.

Two new subcommands are added to `bart` to sign a manifest and to verify a manifest signature.

`bart sign -c` *certificate* `[-k` *key*`]` *manifest*

`bart verify` *manifest*

These commands apply only to XML-format manifests. Manifests created by `bart` will have a single signed section containing all manifest entries. However, if a manifest contains multiple signed sections, `bart verify` will check all of the signatures. This allows individually-generated manifests to be concatenated together to form a single large manifest, if desired.

6.2 Manifest Verification

To allow for on-demand validation of a set of files, a new bart verify subcommand is added with the following syntax:

```
bart validate [-r rules] [-s server]
```

The `validate` operation is similar to the existing `bart create` in that it traverses the filesystem as specified in the rules file. The `validate` operation produces a manifest containing those files that cannot be validated successfully. The administrator can subsequently sign the resulting manifest to use it as input to `signedexecd` if appropriate.

Without the `-s` option, `bart validate` uses open with `O_VERIFY` to validate the files. This shows the administrator what errors will result if an attempt is made to validate the files enumerated by the rules file.

When the `-s` option is specified, `bart validate` uses the specified SignaCert server as a reference instead of the running system. This allows for verification against SignaCert's multi-platform database of file contents.

7 References

1. Trusted Computing Group, Trusted Platform Module Specification, http://www.trustedcomputinggroup.org/specs/TPM/
2. Trusted Computing Group, Reference Manifest Schema Specification, http://www.trustedcomputinggroup.org/specs/IWG/Reference_Manifest_Schema_Specification_v1.0.pdf
3. Trusted Computing Group, TCG PC Client Specific Implementation Specification for Conventional BIOS, http://www.trustedcomputinggroup.org/specs/PCClient/TCG_PCClientImplementationforBIOS_1-20_1-00.pdf
4. Microsoft Corporation, BitLocker Drive Encryption Technical Overview, http://technet2.microsoft.com/WindowsVista/en/library/ba1a3800-ce29-4f09-89ef-65bce923cdb51033.mspx