dm-verity

Device-Mapper's "verity" target provides transparent integrity checking of block devices using a cryptographic digest provided by the kernel crypto API. This target is read-only.

Construction Parameters

```
<version> <dev> <hash_dev>
<data_block_size> <hash_block_size>
<num_data_blocks> <hash_start_block>
<algorithm> <digest> <salt>
[<#opt_params> <opt_params>]
```

<version>

This is the type of the on-disk hash format.

0 is the original format used in the Chromium OS.
The salt is appended when hashing, digests are stored continuously and the rest of the block is padded with zeroes.

1 is the current format that should be used for new devices.
The salt is prepended when hashing and each digest is padded with zeroes to the power of two.

<dev>

This is the device containing data, the integrity of which needs to be checked. It may be specified as a path, like /dev/sdaX, or a device number, <major>:<minor>.

<hash_dev>

This is the device that supplies the hash tree data. It may be specified similarly to the device path and may be the same device. If the same device is used, the hash_start should be outside the configured dm-verity device.

<data_block_size>

The block size on a data device in bytes. Each block corresponds to one digest on the hash device.

<hash_block_size>

The size of a hash block in bytes.

<num_data_blocks>

The number of data blocks on the data device. Additional blocks are inaccessible. You can place hashes to the same partition as data, in this case hashes are placed after <num_data_blocks>.

<hash_start_block>

This is the offset, in <hash_block_size>-blocks, from the start of hash_dev to the root block of the hash tree.

<algorithm>

The cryptographic hash algorithm used for this device. This should be the name of the algorithm, like "sha1".

<digest>

The hexadecimal encoding of the cryptographic hash of the root hash block and the salt. This hash should be trusted as there is no other authenticity beyond this point.

<salt>

The hexadecimal encoding of the salt value.

<#opt_params>

Number of optional parameters. If there are no optional parameters, the optional parameters section can be skipped or #opt_params can be zero. Otherwise #opt_params is the number of following arguments.

Example of optional parameters section:
1 ignore_corruption

ignore_corruption

Log corrupted blocks, but allow read operations to proceed normally.

restart_on_corruption

Restart the system when a corrupted block is discovered. This option is not compatible with ignore_corruption and requires user space support to avoid restart loops.

panic_on_corruption

Panic the device when a corrupted block is discovered. This option is not compatible with ignore_corruption and restart_on_corruption.

ignore_zero_blocks

Do not verify blocks that are expected to contain zeroes and always return zeroes instead. This may be useful if the partition contains unused blocks that are not guaranteed to contain zeroes.

use_fec_from_device <fec_dev>

Use forward error correction (FEC) to recover from corruption if hash verification fails. Use encoding data from the specified device. This may be the same device where data and hash blocks reside, in which case fec_start must be outside data and hash areas.

If the encoding data covers additional metadata, it must be accessible on the hash device after the hash blocks.

Note: block sizes for data and hash devices must match. Also, if the verity <dev> is encrypted the <fec_dev> should be too.

fec_roots <num>

Number of generator roots. This equals to the number of parity bytes in the encoding data. For example, in RS(M, N) encoding, the number of roots is M-N.

fec_blocks <num>

The number of encoding data blocks on the FEC device. The block size for the FEC device is <data_block_size>.

fec_start <offset>

This is the offset, in <data_block_size> blocks, from the start of the FEC device to the beginning of the encoding data.

check_at_most_once

Verify data blocks only the first time they are read from the data device, rather than every time. This reduces the overhead of dm-verity so that it can be used on systems that are memory and/or CPU constrained. However, it provides a reduced level of security because only offline tampering of the data device's content will be detected, not online tampering.

Hash blocks are still verified each time they are read from the hash device, since verification of hash blocks is less performance critical than data blocks, and a hash block will not be verified any more after all the data blocks it covers have been verified anyway.

root_hash_sig_key_desc <key_description>

This is the description of the USER_KEY that the kernel will lookup to get the pkcs7 signature of the roothash. The pkcs7 signature is used to validate the root hash during the creation of the device mapper block device. Verification of roothash depends on the config DM_VERITY_VERIFY_ROOTHASH_SIG being set in the kernel. The signatures are checked against the builtin trusted keyring by default, or the secondary trusted keyring if DM_VERITY_VERIFY_ROOTHASH_SIG_SECONDARY_KEYRING is set. The secondary trusted keyring includes by default the builtin trusted keyring, and it can also gain new certificates at run time if they are signed by a certificate already in the secondary trusted keyring.

Theory of operation

dm-verity is meant to be set up as part of a verified boot path. This may be anything ranging from a boot using tboot or trustedgrub to just booting from a known-good device (like a USB drive or CD).

When a dm-verity device is configured, it is expected that the caller has been authenticated in some way (cryptographic signatures, etc). After instantiation, all hashes will be verified on-demand during disk access. If they cannot be verified up to the root node of the tree, the root hash, then the I/O will fail. This should detect tampering with any data on the device and the hash data.

Cryptographic hashes are used to assert the integrity of the device on a per-block basis. This allows for a lightweight hash computation on first read into the page cache. Block hashes are stored linearly, aligned to the nearest block size.

If forward error correction (FEC) support is enabled any recovery of corrupted data will be verified using the cryptographic hash of the corresponding data. This is why combining error correction with integrity checking is essential.

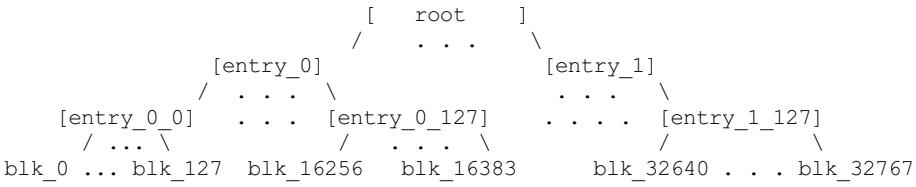
Hash Tree

Each node in the tree is a cryptographic hash. If it is a leaf node, the hash of some data block on disk is calculated. If it is an intermediary node, the hash of a number of child nodes is calculated.

Each entry in the tree is a collection of neighboring nodes that fit in one block. The number is determined based on block_size and the size of the selected cryptographic digest algorithm. The hashes are linearly-ordered in this entry and any unaligned trailing space is ignored but included when calculating the parent node.

The tree looks something like:

alg = sha256, num_blocks = 32768, block_size = 4096



On-disk format

The verity kernel code does not read the verity metadata on-disk header. It only reads the hash blocks which directly follow the header. It is expected that a user-space tool will verify the integrity of the verity header.

Alternatively, the header can be omitted and the dmsetup parameters can be passed via the kernel command-line in a rooted chain of trust where the command-line is verified.

Directly following the header (and with sector number padded to the next hash block boundary) are the hash blocks which are stored a depth at a time (starting from the root), sorted in order of increasing index.

The full specification of kernel parameters and on-disk metadata format is available at the cryptsetup project's wiki page

> https://gitlab.com/cryptsetup/cryptsetup/wikis/DMVerity

Status

V (for Valid) is returned if every check performed so far was valid. If any check failed, C (for Corruption) is returned.

Example

Set up a device:

```
# dmsetup create vroot --readonly --table \
  "0 2097152 verity 1 /dev/sda1 /dev/sda2 4096 4096 262144 1 sha256 "\
  "4392712ba01368efdf14b05c76f9e4df0d53664630b5d48632ed17a137f39076 "\
  "1234000000000000000000000000000000000000000000000000000000000000"
```

A command line tool veritysetup is available to compute or verify the hash tree or activate the kernel device. This is available from the cryptsetup upstream repository https://gitlab.com/cryptsetup/cryptsetup/ (as a libcryptsetup extension).

Create hash on the device:

```
# veritysetup format /dev/sda1 /dev/sda2
...
Root hash: 4392712ba01368efdf14b05c76f9e4df0d53664630b5d48632ed17a137f39076
```

Activate the device:

```
# veritysetup create vroot /dev/sda1 /dev/sda2 \
  4392712ba01368efdf14b05c76f9e4df0d53664630b5d48632ed17a137f39076
```