UBIFS Authentication Support

Introduction

UBIFS utilizes the fscrypt framework to provide confidentiality for file contents and file names. This prevents attacks where an attacker is able to read contents of the filesystem on a single point in time. A classic example is a lost smartphone where the attacker is unable to read personal data stored on the device without the filesystem decryption key.

At the current state, UBIFS encryption however does not prevent attacks where the attacker is able to modify the filesystem contents and the user uses the device afterwards. In such a scenario an attacker can modify filesystem contents arbitrarily without the user noticing. One example is to modify a binary to perform a malicious action when executed [DMC-CBC-ATTACK]. Since most of the filesystem metadata of UBIFS is stored in plain, this makes it fairly easy to swap files and replace their contents.

Other full disk encryption systems like dm-crypt cover all filesystem metadata, which makes such kinds of attacks more complicated, but not impossible. Especially, if the attacker is given access to the device multiple points in time. For dm-crypt and other filesystems that build upon the Linux block IO layer, the dm-integrity or dm-verity subsystems [DM-INTEGRITY, DM-VERITY] can be used to get full data authentication at the block layer. These can also be combined with dm-crypt [CRYPTSETUP2].

This document describes an approach to get file contents _and_ full metadata authentication for UBIFS. Since UBIFS uses fscrypt for file contents and file name encryption, the authentication system could be tied into fscrypt such that existing features like key derivation can be utilized. It should however also be possible to use UBIFS authentication without using encryption.

MTD, UBI & UBIFS

On Linux, the MTD (Memory Technology Devices) subsystem provides a uniform interface to access raw flash devices. One of the more prominent subsystems that work on top of MTD is UBI (Unsorted Block Images). It provides volume management for flash devices and is thus somewhat similar to LVM for block devices. In addition, it deals with flash-specific wear-leveling and transparent I/O error handling. UBI offers logical erase blocks (LEBs) to the layers on top of it and maps them transparently to physical erase blocks (PEBs) on the flash.

UBIFS is a filesystem for raw flash which operates on top of UBI. Thus, wear leveling and some flash specifics are left to UBI, while UBIFS focuses on scalability, performance and recoverability.

```
+------------+ +*******+ +-----------+ +-----+
|            | * UBIFS * | UBI-BLOCK | | ... |
| JFFS/JFFS2 | +*******+ +-----------+ +-----+
|            | +-----------------------------+ +-----------+ +-----+
|            | |            UBI              | | MTD-BLOCK | | ... |
+------------+ +-----------------------------+ +-----------+ +-----+
+------------------------------------------------------------------+
|                  MEMORY TECHNOLOGY DEVICES (MTD)                 |
+------------------------------------------------------------------+
+-----------------------------+ +--------------------------+ +-----+
|         NAND DRIVERS        | |        NOR DRIVERS       | | ... |
+-----------------------------+ +--------------------------+ +-----+

      Figure 1: Linux kernel subsystems for dealing with raw flash
```

Internally, UBIFS maintains multiple data structures which are persisted on the flash:

- *Index*: an on-flash B+ tree where the leaf nodes contain filesystem data
- *Journal*: an additional data structure to collect FS changes before updating the on-flash index and reduce flash wear.
- *Tree Node Cache (TNC)*: an in-memory B+ tree that reflects the current FS state to avoid frequent flash reads. It is basically the in-memory representation of the index, but contains additional attributes.
- *LEB property tree (LPT)*: an on-flash B+ tree for free space accounting per UBI LEB.

In the remainder of this section we will cover the on-flash UBIFS data structures in more detail. The TNC is of less importance here since it is never persisted onto the flash directly. More details on UBIFS can also be found in [UBIFS-WP].

UBIFS Index & Tree Node Cache

Basic on-flash UBIFS entities are called *nodes*. UBIFS knows different types of nodes. Eg. data nodes (`struct ubifs_data_node`) which store chunks of file contents or inode nodes (`struct ubifs_ino_node`) which represent VFS inodes. Almost all types of nodes share a common header (`ubifs_ch`) containing basic information like node type, node length, a sequence number, etc. (see `fs/ubifs/ubifs-media.h` in kernel source). Exceptions are entries of the LPT and some less important node types like padding nodes which are used to pad unusable content at the end of LEBs.

To avoid re-writing the whole B+ tree on every single change, it is implemented as *wandering tree*, where only the changed nodes are re-written and previous versions of them are obsoleted without erasing them right away. As a result, the index is not stored in a single place on the flash, but *wanders* around and there are obsolete parts on the flash as long as the LEB containing them is not reused by UBIFS. To find the most recent version of the index, UBIFS stores a special node called *master node* into UBI LEB 1 which always points to the most recent root node of the UBIFS index. For recoverability, the master node is additionally duplicated to LEB 2. Mounting UBIFS is thus a simple read of LEB 1 and 2 to get the current master node and from there get the location of the most recent on-flash index.

The TNC is the in-memory representation of the on-flash index. It contains some additional runtime attributes per node which are not persisted. One of these is a dirty-flag which marks nodes that have to be persisted the next time the index is written onto the flash. The TNC acts as a write-back cache and all modifications of the on-flash index are done through the TNC. Like other caches, the TNC does not have to mirror the full index into memory, but reads parts of it from flash whenever needed. A *commit* is the UBIFS operation of updating the on-flash filesystem structures like the index. On every commit, the TNC nodes marked as dirty are written to the flash to update the persisted index.

Journal

To avoid wearing out the flash, the index is only persisted (*commited*) when certain conditions are met (eg. `fsync(2)`). The journal is used to record any changes (in form of inode nodes, data nodes etc.) between commits of the index. During mount, the journal is read from the flash and replayed onto the TNC (which will be created on-demand from the on-flash index).

UBIFS reserves a bunch of LEBs just for the journal called *log area*. The amount of log area LEBs is configured on filesystem creation (using `mkfs.ubifs`) and stored in the superblock node. The log area contains only two types of nodes: *reference nodes* and *commit start nodes*. A commit start node is written whenever an index commit is performed. Reference nodes are written on every journal update. Each reference node points to the position of other nodes (inode nodes, data nodes etc.) on the flash that are part of this journal entry. These nodes are called *buds* and describe the actual filesystem changes including their data.

The log area is maintained as a ring. Whenever the journal is almost full, a commit is initiated. This also writes a commit start node so that during mount, UBIFS will seek for the most recent commit start node and just replay every reference node after that. Every reference node before the commit start node will be ignored as they are already part of the on-flash index.

When writing a journal entry, UBIFS first ensures that enough space is available to write the reference node and buds part of this entry. Then, the reference node is written and afterwards the buds describing the file changes. On replay, UBIFS will record every reference node and inspect the location of the referenced LEBs to discover the buds. If these are corrupt or missing, UBIFS will attempt to recover them by re-reading the LEB. This is however only done for the last referenced LEB of the journal. Only this can become corrupt because of a power cut. If the recovery fails, UBIFS will not mount. An error for every other LEB will directly cause UBIFS to fail the mount operation.

```
| ----    LOG AREA     ---- | ----------    MAIN AREA    ------------ |

 -----+------+-----+--------+----   ------+-----+-----+---------------
 \    |      |     |        |   /  /      |     |     |               \
 / CS |  REF | REF |        |   \  \ DENT | INO | INO |               /
 \    |      |     |        |   /  /      |     |     |               \
  ----+------+-----+--------+---   -------+-----+-----+----------------
          |     |                  ^            ^
          |     |                  |            |
          +------------------------+            |
                |                               |
                +-------------------------------+
```

Figure 2: UBIFS flash layout of log area with commit start nodes (CS) and reference nodes (REF) pointing to main area containing their buds

LEB Property Tree/Table

The LEB property tree is used to store per-LEB information. This includes the LEB type and amount of free and *dirty* (old, obsolete content) space [1] on the LEB. The type is important, because UBIFS never mixes index nodes with data nodes on a single LEB and thus each LEB has a specific purpose. This again is useful for free space calculations. See [UBIFS-WP] for more details.

The LEB property tree again is a B+ tree, but it is much smaller than the index. Due to its smaller size it is always written as one chunk on every commit. Thus, saving the LPT is an atomic operation.

[1] Since LEBs can only be appended and never overwritten, there is a difference between free space ie. the remaining space left on the LEB to be written to without erasing it and previously written content that is obsolete but can't be overwritten without erasing the full LEB.

UBIFS Authentication

This chapter introduces UBIFS authentication which enables UBIFS to verify the authenticity and integrity of metadata and file contents stored on flash.

Threat Model

UBIFS authentication enables detection of offline data modification. While it does not prevent it, it enables (trusted) code to check the integrity and authenticity of on-flash file contents and filesystem metadata. This covers attacks where file contents are swapped.

UBIFS authentication will not protect against rollback of full flash contents. Ie. an attacker can still dump the flash and restore it at a later time without detection. It will also not protect against partial rollback of individual index commits. That means that an attacker is able to partially undo changes. This is possible because UBIFS does not immediately overwrites obsolete versions of the index tree or the journal, but instead marks them as obsolete and garbage collection erases them at a later time. An attacker can use this by erasing parts of the current tree and restoring old versions that are still on the flash and have not yet been erased. This is possible, because every commit will always write a new version of the index root node and the master node without overwriting the previous version. This is further helped by the wear-leveling operations of UBI which copies contents from one physical eraseblock to another and does not atomically erase the first eraseblock.

UBIFS authentication does not cover attacks where an attacker is able to execute code on the device after the authentication key was provided. Additional measures like secure boot and trusted boot have to be taken to ensure that only trusted code is executed on a device.

Authentication

To be able to fully trust data read from flash, all UBIFS data structures stored on flash are authenticated. That is:

- The index which includes file contents, file metadata like extended attributes, file length etc.
- The journal which also contains file contents and metadata by recording changes to the filesystem
- The LPT which stores UBI LEB metadata which UBIFS uses for free space accounting

Index Authentication

Through UBIFS' concept of a wandering tree, it already takes care of only updating and persisting changed parts from leaf node up to the root node of the full B+ tree. This enables us to augment the index nodes of the tree with a hash over each node's child nodes. As a result, the index basically also a Merkle tree. Since the leaf nodes of the index contain the actual filesystem data, the hashes of their parent index nodes thus cover all the file contents and file metadata. When a file changes, the UBIFS index is updated accordingly from the leaf nodes up to the root node including the master node. This process can be hooked to recompute the hash only for each changed node at the same time. Whenever a file is read, UBIFS can verify the hashes from each leaf node up to the root node to ensure the node's integrity.

To ensure the authenticity of the whole index, the UBIFS master node stores a keyed hash (HMAC) over its own contents and a hash of the root node of the index tree. As mentioned above, the master node is always written to the flash whenever the index is persisted (ie. on index commit).

Using this approach only UBIFS index nodes and the master node are changed to include a hash. All other types of nodes will remain unchanged. This reduces the storage overhead which is precious for users of UBIFS (ie. embedded devices).

```
                      +---------------+
                      |  Master Node  |
                      |    (hash)     |
                      +---------------+
                              |
                              v
                     +-------------------+
                     |  Index Node #1    |
                     |                   |
                     | branch0   branchn |
                     | (hash)    (hash)  |
                     +-------------------+
                        |    ...   |  (fanout: 8)
                        |          |
                +-------+          +------+
                |                         |
                v                         v
      +-------------------+       +-------------------+
      |  Index Node #2    |       |  Index Node #3    |
      |                   |       |                   |
      | branch0   branchn |       | branch0   branchn |
      | (hash)    (hash)  |       | (hash)    (hash)  |
      +-------------------+       +-------------------+
           |   ...                     |   ...   |
           v                           v         v
         +-----------+         +----------+  +-----------+
         | Data Node |         | INO Node |  | DENT Node |
         +-----------+         +----------+  +-----------+


   Figure 3: Coverage areas of index node hash and master node HMAC
```

The most important part for robustness and power-cut safety is to atomically persist the hash and file contents. Here the existing UBIFS logic for how changed nodes are persisted is already designed for this purpose such that UBIFS can safely recover if a power-cut occurs while persisting. Adding hashes to index nodes does not change this since each hash will be persisted atomically together with its respective node.

Journal Authentication

The journal is authenticated too. Since the journal is continuously written it is necessary to also add authentication information frequently to the journal so that in case of a powercut not too much data can't be authenticated. This is done by creating a continuous hash beginning from the commit start node over the previous reference nodes, the current reference node, and the bud nodes. From time to time whenever it is suitable authentication nodes are added between the bud nodes. This new node type contains a HMAC over the current state of the hash chain. That way a journal can be authenticated up to the last authentication node. The tail of the journal which may not have a authentication node cannot be authenticated and is skipped during journal replay.

We get this picture for journal authentication:

```
    ,,,,,,,,
    ,......,...........................................
    ,. CS  ,               hash1.----.           hash2.----.
    ,.  |  ,                    .    |hmac            .    |hmac
    ,.  v  ,                    .    v                .    v
    ,.REF#0,-> bud -> bud -> bud.-> auth -> bud -> bud.-> auth ...
    ,..|...,...........................................
    ,  |   ,
    ,  |   ,,,,,,,,,,,,,,,
    .  |            hash3,----.
    ,  |                 ,    |hmac
    ,  v                 ,    v
    , REF#1 -> bud -> bud,-> auth ...
    ,,,|,,,,,,,,,,,,,,,,,,
       v
      REF#2 -> ...
       |
       V
      ...
```

Since the hash also includes the reference nodes an attacker cannot reorder or skip any journal heads for replay. An attacker can only remove bud nodes or reference nodes from the end of the journal, effectively rewinding the filesystem at maximum back to the last commit.

The location of the log area is stored in the master node. Since the master node is authenticated with a HMAC as described above, it is not possible to tamper with that without detection. The size of the log area is specified when the filesystem is created using *mkfs.ubifs* and stored in the superblock node. To avoid tampering with this and other values stored there, a HMAC is added to the superblock struct. The superblock node is stored in LEB 0 and is only modified on feature flag or similar changes, but never on file changes.

LPT Authentication

The location of the LPT root node on the flash is stored in the UBIFS master node. Since the LPT is written and read atomically on every commit, there is no need to authenticate individual nodes of the tree. It suffices to protect the integrity of the full LPT by a simple hash stored in the master node. Since the master node itself is authenticated, the LPTs authenticity can be verified by verifying the authenticity of the master node and comparing the LTP hash stored there with the hash computed from the read on-flash LPT.

Key Management

For simplicity, UBIFS authentication uses a single key to compute the HMACs of superblock, master, commit start and reference nodes. This key has to be available on creation of the filesystem (*mkfs.ubifs*) to authenticate the superblock node. Further, it has to be available on mount of the filesystem to verify authenticated nodes and generate new HMACs for changes.

UBIFS authentication is intended to operate side-by-side with UBIFS encryption (fscrypt) to provide confidentiality and authenticity. Since UBIFS encryption has a different approach of encryption policies per directory, there can be multiple fscrypt master keys and there might be folders without encryption. UBIFS authentication on the other hand has an all-or-nothing approach in the sense that it either authenticates everything of the filesystem or nothing. Because of this and because UBIFS authentication should also be usable without encryption, it does not share the same master key with fscrypt, but manages a dedicated authentication key.

The API for providing the authentication key has yet to be defined, but the key can eg. be provided by userspace through a keyring similar to the way it is currently done in fscrypt. It should however be noted that the current fscrypt approach has shown its flaws and the userspace API will eventually change [FSCRYPT-POLICY2].

Nevertheless, it will be possible for a user to provide a single passphrase or key in userspace that covers UBIFS authentication and encryption. This can be solved by the corresponding userspace tools which derive a second key for authentication in addition to the derived fscrypt master key used for encryption.

To be able to check if the proper key is available on mount, the UBIFS superblock node will additionally store a hash of the authentication key. This approach is similar to the approach proposed for fscrypt encryption policy v2 [FSCRYPT-POLICY2].

Future Extensions

In certain cases where a vendor wants to provide an authenticated filesystem image to customers, it should be possible to do so without sharing the secret UBIFS authentication key. Instead, in addition the each HMAC a digital signature could be stored where the vendor shares the public key alongside the filesystem image. In case this filesystem has to be modified afterwards, UBIFS can exchange all digital signatures with HMACs on first mount similar to the way the IMA/EVM subsystem deals with such situations. The HMAC key will then have to be provided beforehand in the normal way.

References

[CRYPTSETUP2] https://www.saout.de/pipermail/dm-crypt/2017-November/005745.html

[DMC-CBC-ATTACK] https://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/

[DM-INTEGRITY] https://www.kernel.org/doc/Documentation/device-mapper/dm-integrity.rst

[DM-VERITY] https://www.kernel.org/doc/Documentation/device-mapper/verity.rst

[FSCRYPT-POLICY2] https://www.spinics.net/lists/linux-ext4/msg58710.html

[UBIFS-WP] http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf