http://www.opensolaris.org/os/project/zfs-crypto/files/zfs-crypto-design.pdf

ZFS Encryption Project

Darren J Moffat

2008-01-27

It is assumed that the reader of this document is familiar with basic ZFS administration concepts and the high level ZFS architecture.

1 Overview

What are we doing ?

This project will provide on disk encryption/decryption support for ZFS datasets. The project will cover the addition of encryption and decryption to the ZFS IO pipeline and the key management for enctyped ZFS datasets.

It will deliver in multiple phases to support different key management strategies including one which provides support for secure deletion based on encrypted datasets.

Why are we doing this ?

- Untrusted path to SAN: We trust them to deliver on the service level agreement but not with the clear text of our data.
- Theft of physical storage: Racks or disks from a datacentre or a lost laptop
- Secure deletion: forget the key and everything is gone.
- Allow for mix of cleartext and encrypted data within a storage pool, full disk encryption doesn't allow for this.

How are we doing this ?

The ZFS on disk versioning system will be used to ensure pool compatibility when this feature is introduced. The raw encryption/decryption functionality will only be implemented in the kernel .

2 Core Requirements

2.1 What must be encrypted?

All data and file system metadata, such as file owner ACL size etc, must be encrypted when on disk. For ZVOLs all data is encrypted on disk.

2.2 What can be in clear?

ZFS level metadata such as dataset properties (including user defined properties), dataset (not file) ACLs for user delgation, dataset names.

2.3 When is clear text available?

Clear text data must only be available to applications when the filesystem is mounted and the correct key material presented. Clear text will be available in the ZFS in memory ARC. Data in the L2ARC is encrypted since the L2ARC may be using storage devices that persist data on power off.

2.4 Existing pools

Existing pools must be able to upgrade their version to gain the ability to contain encrypted data.

2.5 Pool Scrub / Resilver

To ensure pool integrity at all times resilvering of vdevs and pool integrity check (scrub) must be able to be performed without any key material present. The ZIL claim phase must not require keys be present.

2.6 Encryption Policy control

The system must allow for both encrypted and clear text data in the same storage pool.

File systems (datasets are either file systems or ZVOLs) are relatively very cheap in ZFS compared with traditional filesystems. Some other systems allow encryption policy to be set at the folder/directory level. Using the dataset in ZFS provides equivalent functionality.

Encryption policy covers: the algorithm, the keylength and the algorithm mode. The initial release supports only AES for encrypting data but the project must provide a method to add new algorithms/modes/keylengths.

While it is probably desirable not to expose the algorithm mode to the end user/administrator (most won't know the difference between the unsuitable CBC/EBC modes and the choosen CCM mode) we choose to do so to ensure extensibility based on future modes while keeping the same base algorithm, for example we deliver initially with support for AES CCM, future research could either find flaws in CCM unknown today or lead to an encryption mode more suitable for the job.

2.7 Why not full disk (vdev) encryption ?

While it would be technically possible to provide an encryption functionality at the ZFS vdev rather than dataset layer it was deemed unsuitable since it would not allow for a mix of clear and encrypted datasets nor would it allow for datasets to be encrypted with different algorithms/keylengths. A vdev layer encryption would also require that encryption was set only at pool creation time. Encryption at the vdev layer would require keys to be present for resilver/scrub.

2.8 Key storage

2.8.1 Full Hardware

It must be possible to have all cryptographic key material stored and/or created entirely in a hardware device, such as the SCA-6000, when one is available to the system.

2.8.2 Software only

It must also be possible to have all of the functionality without requiring hardware; the only difference should be the security of the key material and the performance (many hardware keystores also accelerate cryptographic algorithms such that they are faster than software)

2.8.3 Mixed mode

It must also be possible to operate in a mixed mode where the most valuable keys are stored in hardware when the power is off but due to the slow performance of intermittent availability of the hardware keystore the encryption of data is performed in software.

A possible example of this case is using a Smartcard or TPM chip as the store for the master keys, typically these are slow and have a small amount of key storage space compared to an device such as the SCA-6000.

2.9 Alternate cryptographic algorithms/modes

Initially only AES will be supported with key lengths of 128, 192 and 256 using the CCM. The design and implementation must allow for alternate algorithms and modes. It does not need to provide this via a plugin system, but minimal code change should be required to add a new algorithm/mode.

2.10 Encrypted ZFS backups

The existing zfs(1) send and receive options must continue to work. Only clear text data stream will be supported.

Existing backup programs that use POSIX interfaces must continue to work.

2.11 User /Zonedelegation of encryption functionality

The ability to delegateto users/groups and zones the ablity to turn on/off encryption and the management and use of the encryption keys must be provided.

3 High Level Architecture

3.1 Key Management

Encryption property, algorithm and key source are set at the ZFS dataset level. Each dataset has a unique (randomly generated) key created at dataset creation time. The algorithm mode and key length are fixed at dataset creation time. The randomly generated data encryption key is stored on disk with the dataset in wrapped form, see "Key Wrapping" section below. It is wrapped by a master per pool key or per dataset key that the user/admin provides. The use of wrapped keys allows us to support alternate key management systems in the future, including remote key management stations.

3.1.1 Key change

Using per dataset wrapped keys allows for a key change system that does not require all encrypted blocks to be read and decrypted then re-encrypted and written using the new key. The only keys that are changed are those that the admin/user manages and provides to ZFS, the actual encryption keys are not changed.

3.2 Secured dataset deletion

If the master pool key is changed after a dataset is deleted the the old wrapped key property (now on the free list) can no longer be decrypted with the new user provided key. This provides a simple whole dataset secure deletion by key destruction mechanism without requiring a "secure" overwrite of the blocks on the free list.

3.3 Snapshots/Clones

Snapshots of datasets are always encrypted if their parent is encrypted.

Since clones reference data from their parent dataset they are also always encrypted if their parent was encrypted. Clones use the same data encryption key as their parent but may have a different wrapping key.

3.4 Encrypted ZVOL

Since ZVOLS are just datasets that are read and written via the ZIO pipeline they too will benefit from encryption. When it becomes possible to use a ZVOL as system swap it should be possible to swap on an encrypted ZVOL. Using an encrypted ZVOL for crash dumps is may not be advisable, when it becomes

possible to use a ZVOL as the system dump device checks should be added to ensure that ZVOL is not encrypted or that dumping to an encrypted ZVOL is reliable.

3.5 User Delegation & Zones

In addition to delegation of the new per dataset properties two additional delegations will be provided. The **keyuse** delegation controls key load/unload. The **keychange** delegation controls wrapping key change. Using two separate delegations allows for delegation of key use without key change, thus providing the possibility for an external (to ZFS) key escrow system. The **keyuse** and **keychange** delegations are in addition to the delegations provided by the properties that define which key material to use.

ZFS Filesystem administrators in non-global zones may only enable/use encryption on a datasets delegated to that zone even if the user would otherwise have the above delegations.

4 Prerequsisites

4.1 ZFS

4.1.1 Data set creation time options

This is needed to ensure that all data in a dataset was always encrypted. Once set the encryption property can not be turned off. This functionality is currently available for all datasets except the top level dataset that shares the pool name. Until this issue is resolved the top level dataset can not benefit from encryption. If the desire is that all datasets are encrypted the recommended workaround is to set the “canmount” property on the top level after pool creation to false.

4.1.2 ZIL refactor

The way that ZIL records are written to disk needs to be changed so that during initial pool import (eg on reboot after crash) the claim of blocks used by the ZIL can be performed even if the required key for the dataset is unavailable. The decryption of the data and resubmission to the write queque will only be done when the dataset is mounted (ie the key is present).

4.2 Cryptographic Framework

The only additional functionality from the cryptographic framework required is a software implementation of AES CCM and AES CBC_PAD in kernel. Each hardware provider, such as the SCA-6000, will also need to provide AES CCM to work with ZFS for acceleration and/or keystore. If a future crypto framework project provides CCM support at the framework level the requirement of each

provider to do this may be relaxed, however this could have implications for running providers in FIPS 140-2 mode (further discussion of this issue is not covered in this document).

5 Detailed Design

5.1 Import/Export & Mount/Umount

Datasets that are encrypted will only be mounted at pool import time if the necessary key material is available. Importing a pool will not cause any interaction to retrieve key material, this is important since pool import may happen during very early system boot.

Similarly a 'zfs mount -a' will skip over those datasets who's keys are not available. A direct mount of a single filesystem requires that its key already have been made available.

A dataset key maybe unloaded while it is mounted, in this case some data may be unencrypted in the ARC and could still be passed to applications. Read requests for data not in the ARC will fail with permission denied. Write requests will fail as defined by the pool level "failmode" property and may continue if the key is later provided.

Exporting a pool will remove from system memory all unwrapped dataset keys and all loaded wrapping keys in the control of ZFS. All unencrypted data store in the ZFS ARC will be removed from memory.

5.2 Dataset & Pool Properties/Values

All of the following properties can be delegated using the existing property delegation system.

5.2.1 Dataset: encryption

A new set once property "encryption" will be added. It will not be an editable value after the filesystem/ZVOL has been created since all data in a dataset must be encrypted with the same algorithm/keylength/mode and the same key value. The values will be *"on | off | aes-128-ccm | aes-256-ccm"*, with *aes-256-ccm* being the *"on"* value. Unless set or inherited the default is off.

5.2.2 Dataset: checksum

A new checksum type of *"sha256+ccm"* is added. In this checksum mode the SHA256 checksum is truncated to 128 bits and stored in the second two words of the ZFS checksum. The first to words of the checksum are used to store the AES CCM auth tag. The checksum *"sha256+ccm"* can not be set directly and is automatically set if encryption is turned on (and using ccm mode). When

encryption is enabled for a dataset checksum becomes a read-only property.

5.3 Dataset/Pool: keysource

A read/write property giving the location of the wrapping key. This is not the actual key but a "pointer" on how to retrieve it. If encryption is off this property defaults to "-", however it can be set to allow for it to be inherited from a parent dataset that is not encrypted. This property must be set or inherited if encryption is on for a dataset. Note that is property has values that may cause 'zfs create' and 'zfs key -l' (see below) to interactively prompt the user for input. All functionality is supported in prompted and non prompted mode though to ensure sufficient flexibility for scripting and interactive use. Examples are provided in the draft man pages accompanying this document.

The property syntax is as follows:

keysource=<format>,<locator>

<format>: *raw* | *hex* | *passphrase* | *token*

raw:

The key is a binary 128 or 256 bits

hex:

As raw but encoded in hex, this is provided to allow for easy cut/paste

p*assphrase:*

The key is a passphrase up to 255 UTF-8 characters in length. The passphrase is not used directly but passed through PKCS#5 PBE to generate the actual encryption key. The PBE algorithm is salted using the per dataset guid.

token:

The key is stored in some external keystore system as defined in the locator field. It is assumed that the key can not be removed from the token and can only be used by reference. The only locator value supported for token is the pkcs11 URI.

<locator>: *prompt* | *URI*

prompt:

The ZFS CLI will use stdin with echo off to retrieve the key in the appropriate format. When the format is passphrase and the key is being given during creation time or the key is being changed the user will be required to enter it twice this will not be required for the other format types. Similar behaviour will be used in the ZFS web based GUI.

f*ile://*

The key (in raw, hex or a passphrase) is located at the file URI. It is strongly advised that the file:// URI point to a file on removable media and not to a file elsewhere in the same storage pool. This will not be enforced though since due to automouner and intra zone lofs mounts it could be unreliable.

pkcs11://

Since this URI standard is not yet defined integration of this locator type will not be included in this phase of the project. It is included here only to show the intent, the exact syntax and semantics will be the subject of a separate review. Since this is the only token URI defined the token format will not integrate until the pkcs11: URI support integrates. It is expected to look similar to the following examples:

pkcs11://token=”Bobs Token”;label=”My Key”;

Future projects may add other URI types such as https://

The value of the keysource property maybe changed without doing a key change, this allows for changing the location of the key without changing its value. Not all transitions are value however. Changing from raw/hex to passphrase is not allowed. Chaning the locator from prompt to file:// or changing the file:// URI while keeping the same format is acceptable.

The key is not checked for presence if the property value is being changed directly with the zfs/zpool set command. This allows for changing the value of the property with the key is not present.

If administrators do not wish users to change the per dataset keysource property it can be denyed from delegation while still allowing users to enable/disable encryption on a dataset and allowing them to load/change the key value. One possible use of this is forcing users to use a prompted passphrase rather than a key in a file.

5.3.1 Dataset: keyscope

A keyscope index property will also be added. This indicates where to look for the keysource property to find the wrapping key used for this dataset. For the phase 1 delivery only two values will be allowed “*pool* | *dataset* ”.

If not specified (or inherited) and encryption is on for a dataset the default keyscope is “pool”. When encryption=off keyscope defaults to the “-” value. Follow on projects that add more key management types may choose to do so by providing more values for this property.

The *pool* key-mode is the default for any dataset that has the encryption property set to an on value. In this mode we the randomly generated per dataset keys are wrapped with the per pool key that is defined in the pool keysource property.

When keyscope is *dataset* the per dataset **keysource** property must also be set (or inherited).

It is possible to transition between *pool* and *dataset* keyscope only by initiating a key change (see below), the property can not be set directly with the zfs command.

5.3.2 Dataset: keystatus

A non persistent index property **keystatus** provides the ability to determine if the required key for a given dataset is present. The index values are "*available* | *unavailable*". For datasets with encryption=off the **keystatus** value is not applicable and querying it returns "-". This property is used internally by the share and mount commands as well as being visible for external scripting use.

5.3.3 Pool: keysource

For **keyscope** of **pool** the **keysource** property on the pool is used instead of the per dataset property. The per dataset **keysource** property has the same syntax as the per dataset **keysource**.

5.3.4 Pool: keystatus

Similar to the per dataset keystatus property. The same values *"available* | *unavailable*" are used. If the pool level keysource property is not set the the pool **keystatus** property returns "-".

5.4 zfs/zpool key command

A new command is added to both zpool(1M) and zfs(1). The **key** command is responsible for the actually load and unload of the keys, and key change. Setting the keyscope or keysource properties with zfs set does not cause a key to be loaded only to set the policy for where the key is stored.

key provides the following options:

-l: Load the key defined in **keysource**. For zfs(1) this also mounts the named dataset. For zpool(1) all datasets with **keyscope** of pool are mounted.

-u: unload the dataset or pool key from memory. It does not unmount the dataset(s).

-c: change the wrapping key value. This operation allows the user to provide a new wrapping key. The dataset(s) remain mounted and available during the key change (since no data is actually reencrypted). The old key must already have been loaded before a change can be performed. If the keysource has a format passphrase then the user will be prompted twice for the new key.

-o: The keysource property may also be changed during a key change operation. If -o is given then the keysource property is updated if the change was successful. Providing -o keysource is the only way to change between a passphrase and raw/hex key.

5.5 ZFS GUI

The ZFS GUI will be able to set/view the necessary properties and provide for creating encrypted datasets. This will be provided via extensions to the private APIs in the existing libzfs/libzfs_jni libraries. Early mockups are provided as image files along with this document.

5.6 ZIO changes

A new stage for read and write will be added to the ZIO pipeline. This stage will deal only with encryption and decryption of the data. For write the encryption stage is added after compression but before checksum. For read we decrypt after checksum and before decompression.

In some cases it may be desirable for performance to perform the encryption and checksum operations in a single step. The cryptographic framework has this support. This is an implementation choice and doesn't impact the user interface or on disk layout.

5.7 Interaction with the ZFS cache (ARC)

It is important not to use the OpenSolaris Cryptographic Framework functionality that encrypts the data in place otherwise this will result in the data stored in the ARC being encrypted in addition to the data stored on disk. The result of this is that applications can no longer get access to the clear text data while we continue to have cache hits for those blocks.

Any data for an encrypted dataset that is being written to an L2ARC device will be encrypted. The data could be getting written to an L2ARC vdev when the key for the dataset has been unloaded, but the dataset is still marked as mounted (and thus still has unencrypted data in the in memory cache). For this reason the L2ARC encryption will use an ephemeral key, generated the first time we need to use it, that is never written to disk but stored with the in memory spa structure (not even in wrapped form). This does mean that it is not possible for blocks written to the L2ARC vdevs to be used to prime the cache after a pool export or reboot.

5.8 ZFS on disk Blocks

The dnode_t and dnode_phys_t will gain an additional uint8_t sized type, similar to the existing dn_compression, called dn_crypt. In the case of dnode_phys_t this will be taken out of the reserved space in dn_pad2. Values for dn_crypt will be implemented similar to how this is done for compression. A new zio_encrypt

enumeration will be created with the initial values as defined in the table below. The value of ZIO_CRYPT_ON is ZIO_CRYPT_AES_256_CCM. The default value for encryption is ZIO_CRYPT_OFF.

Value	UI String	checksum
ZIO_CRYPT_INERIT = 0	inherit	
ZIO_CRYPT_ON	on	
ZIO_CRYPT_OFF	off	any
ZIO_CRYPT_AES_128_CCM	aes-128-ccm	sha256+ccm
ZIO_CRYPT_AES_256_CCM	aes-256-ccm	sha256+ccm

Currently some symbolic links are stored in the dn_bonus part of the dnode_phys_t and will thus need special attention to ensure it is encrypted. This actually applies to any data store in the dn_bonus area.

5.9 Key Wrapping

Each dataset that has encryption enabled has its own key randomly generated during dataset creation. This randomly generated key is the one used for the actual encryption of data. The data encryption key is wrapped using AES_CBC with PKCS#7 padding. The dataset guid (a randomly generated uint64_t) combined with a monotonically increasing key version number (uint64_t) are used as the IV. The resulting wrapped (encrypted) key is stored the hidden **wrappedkey** property. The key version number is stored in the hidden **keyvno** property. The key version number is required to ensure that we use a different IV when we change the wrapping key since the dataset guid is constant.

6 Futures

None of the features listed in this section of the document are included in the phase 1 delivery. They are listed in this document only to provide context and an indication of possible features in later phases.

6.1 Encrypted root

While this may be desirable in some cases it is out of scope for the first phase of the ZFS crypto project.

6.2 Migration between unencrypted and encrypted datasets

A solution for migrating between encrypted an unencrypted datasets is desirable in some cases. In an ideal world this would be an online method, however it is unlikely that this will be possible and instead a method using backup/restore will be

recommended. The interim solution will be to use zfs(1) send/recv. The main problem with an online migration is that due to the COW nature of ZFS the old cleartext data would still be available somewhere on disk. There is currently no functionality available in ZFS to do a "secure" overwrite of blocks on the free list; such a feature is a pre-requisite for reinvestigating a capability to encrypt existing datasets.

6.3 Unencrypted filesystem Metadata

A mode with ciphertext file data but cleartext file system (ZPL) metadata may be desirable to allow easy use of unmodified 3rd party backup software without depending on it to encrypt the data over the network or when writing to the backup media. The restore part of this is non trivial to implement and as such this is left out this phase of the project and maybe re-investigated in the future.

7 References

7.1 http://opensolaris.org/os/community/zfs/source/

7.2 http://opensolaris.org/os/community/zfs/docs/ondiskformatfinal.pdf

7.3 RSA PKCS#11 v2.20 http://www.rsasecurity.com

7.4 RFC 3610 Counter with CBC-MAC (CCM)