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

ZFS Encryption Project

Darren J Moffat

2007-07-02

This is a work in progress document of the zfs-crypto project on opensolaris.org NOT a feature commitment from Sun Microsystems for a future release of Solaris.

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 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.

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 and will be isolated to its own source files. This is to assist with ease of porting this functionality to platforms other then OpenSolaris that also have the ZFS filesystem. It is possible that not all ports will have identical functionality with respect to their cryptographic frameworks, however the should all be able to use the same pools the only differences should be in how the cryptographic frameworks operate, for example hardware key storage or acceleration may not be available on all platforms.

2 Core Requirements

2.1 What is encrypted?

All data and file system metadata, such as file owner ACL size etc, must be encrypted when on disk. All dataset user properties will be encrypted.

For ZVOLs all data is encrypted on disk.

2.1.1 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 cache (the ARC).

2.2 Encryption Policy control

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. While it is probably desirable not to expose the algorithm mode to the end administrator we may choose to do so to ensure extensibility for future modes a given algorithm and key length, for example deliver initially with support for AES in CCM mode but later add GCM mode.

The encryption policy will be set at the level of a ZFS dataset. This allows for ZFS pools that contain a mix of cleartext and ciphertext data. A single dataset will always have a consistent policy (i.e. encryption is either on or off). This means that the encryption property can only be set at dataset creation time and can not be later changed. This is unlike the compression property which may be changed at any time.

2.3 Key storage

2.3.1 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. This may limit the choice of encryption and checksum algorithms in some deployments.

2.3.2 Software only

It must also be possible to have all of the functionality of ZFS Cryptographic support without requiring hardware; the only difference should be the security of

the key material and the performance (it is assumed that most hardware keystores also accelerate cryptographic algorithms such that they are faster than software)

2.3.3 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 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 HSM such as the SCA-6000.

2.4 Alternate cryptographic algorithms

Initially only AES will be supported with key lengths of 128 and 256 using the CCM and CBC modes. The design and implementation must allow for alternate algorithms.

2.5 Encrypted ZFS backups

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

3 High Level Architecture

3.1 Key Management

Encryption property, algorithm and key are set at the ZFS dataset level by the zfs command. Each dataset has a unique key generated at creation time. This key is stored on disk wrapped by a master per pool key in the initial phase. The use of wrapped keys allows us to support alternate key management systems in the future, including remote key management stations and per user key access.

[We need to talk about this. Writing the key to the disk, even wrapped can conflict with the need for key destruction (4.1.2) because if you do not change the pool key, then changing or destroying the dataset key will not be effective for any media that has already been lost, stolen, broken, repurposed, etc.]

3.1.1 Key changes

A future general purpose mechanism in ZFS for “updating” blocks as a result of changing compression/checksum/encryption properties will be used as the basis for a solution. This solution will need to be extended for encryption so that we know

when it is safe to destroy the old keys. A per block key version system will be needed.

3.2 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.

3.3 Encrypted ZVOL

Since ZVOLS are just datasets that are read and written via the ZIO pipeline they too will benefit from encryption. When using a ZVOL for the system swap we do not need to, nor is it actually desirable to do so, preserve the key for future uses of the pool on the same or other systems. To address this an additional dataset property will be added to indicate if the keys are ephemeral or persist with the pool.

3.4 Zones & encrypted data sets

ZFS Filesystem administrators in non-global zones may only set the encryption property to an on value if a per pool key has already been set by the global zone administrator and the data set is delegated to that zone.

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.

4.1.2 Option relations

We need a way to force a particular checksum algorithm, eg. SHA256, automatically set when encryption is set. This also means that checksum becomes a property that can't be changed in some cases.

One case were we must force this is when encryption is set to a CBC mode of AES, we must force checksum to be HMAC-SHA256.

4.2 Cryptographic Framework

The only additional functionality from the cryptographic framework required is a software implementation of AES CCM mode. Each hardware provider, such as the SCA-6000, will also need to provide AES CCM or AES CBC to work with ZFS for acceleration and/or keystore.

Need a way to interact with the user during boot or service restart to get the master passphrase or token PIN entered.

5 Detailed Design

5.1 Dataset & Pool Properties

5.1.1 Dataset: encryption

A new property “encryption” will be added. It will not be an editable value after the filesystem has been created. The values will be “on | off | aes-128-ccm | aes-256-ccm | aes-128-cbc | aes-256-cbc”, with aes-128-cbc being the default. The mode is exposed in the property value to ensure that additional modes of AES can be added in the future. When a CBC mode mode is used checksum is set to hmac-sha256 and becomes read-only.

5.1.2 Dataset: checksum

A new checksum type of “hmac-sha256” is added. The HMAC key will be stored in the same way as the encryption

5.1.3 Dataset: key-mode

A key-mode property will also be added. This indicates the key management system in use for this data set. For the phase 1 delivery only two values will be allowed “pwrap | ephemeral” the default is pwrap.

The pwrap key-mode is the default for any dataset that has the encryption property set to an on value. In this mode we randomly generate per dataset keys and store them wrapped with the per pool key.

When the ephemeral key-mode is used we generate a random key each time the dataset is made available and do not store it on disk. This is intended to be used when swapping to a ZVOL, though there may be other possible uses such as for a dataset with /var/tmp like properties.

5.2 Pool: setkey

As mentioned above the dataset keys are randomly generated, this means that creation or mount of a ZFS dataset does not need to prompt for a per dataset key. Instead a master key for the pool is used to wrap the dataset keys.

For some modes of operation we will need a way to interact with the user to enter the master passphrase or the PIN for the hardware device when a pool is brought online.For appliance systems, and some other deployment cases, having the raw key on a USB attached drive and having no interaction with any user or admin is highly desirable. Two modes will be provided for this case, one allows

pointing to the full path of a file containing the key, the other will search the root directory of all mounted removable media filesystems for a file named using the pool GUID in a specifed directory. Other parts of the appliance system would be responsible for ensuring the media containing the raw key is properly ejected and/or otherwise not available. It is also assumed that there are physical security procedures to ensure the removable drives containing the key material are not left permanently connected.

5.3 ZFS GUI

The ZFS GUI will be able to set the encryption and key-mode properties using the existing libzfs/libzfs_jni interfaces. The new per pool setkey functionality will be implemented as an API in libzfs and libzfs_jni. The command line interface interaction with the admin will be implemented in the zpool command.

5.4 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.

5.5 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.

5.6 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_128_CBC. CBC mode is chosen as the default to increase the chances of using hardware acceleration since most hardware does not support CCM mode at this time. 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	any
ZIO_CRYPT_AES_256_CCM	aes-256-ccm	any
ZIO_CRYPT_AES_128_CBC	aes-128-cbc	hmac-sha256
ZIO_CRYPT_AES_256_CBC	aes-256-cbc	hmac-sha256

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 or deprecated from use. This actually applies to any data store in the dn_bonus area.

6 Future Features

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 Secure deletion

Traditional secure deletion has involved attempting to reduce the likely hood of the data been read back from the disk platters by overwritting the cleartext data with random patterns and or zeros.

When a filesystem is encrypted we can ensure the data is securely deleted when there are no online copies of the data (ie the dataset is not mounted) and the master key used to encrypt that data set has been destroyed.

While this still leaves us with the problem of securely destroying a key it is a much simpler problem than destroying potentially terabytes of data, it is also likely to be much quicker.

If the master key is stored in a removable device such as a smartcard or a PCI card it maybe possible that physical destruction of that card (and all the clones of it) is sufficient and timely – for example dropping the smartcard into an acid bath.

The actually means of key destruction are not included in this document and are left as a mater of policy for invidual deployments.

6.1.1 On Demand

From the users view point this is the simplest form of secure deletion, ie make it happen now.

6.1.2 Timed

This is an automatic form of secure deletion. Each dataset would use a key for a specific data expiry time. The key manager will deal with destroying the key at the agreed time.

6.2 Encrypted root

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

6.3 Migration between unencrypted and encrypted datasets

A solution for migrating between encrypted an unencrypted datasets needs to be provided. 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.

A similar solution to that used for key change will be able to provide in place migration between cleartext and ciphertext in the future.

6.4 Unencrypted 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.

This may mean encrypting DMU_OT_PLAIN_FILE_CONTENTS only.

6.5 Split data & metadata keying

An alternate and more secure method to implementing the 3rd party backup solution outlined above may be to use different key material for file data and POSIX filesystem metadata. This would allow backup software to present only the metadata key.

7 References

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

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