Steg: a deniably-encrypted block device

Leopold Samulis

26/11/2011

1 What is Steg / DM-Steg?

Steg is a specification for deniably-encrypted block devices

DM-Steg is an implementation of this specification for (GNU/)Linux

Steg works with substrates (devices containing ciphertext) to export plaintext-containing block devices, known as aspects, to the user. Without having the key(s), there is no way of determining how many aspects a substrate contains, or if it contains any aspects at all. Aspects, being block devices, may also function as substrates for the purpose of creating aspects-within-aspects. Steg imposes no limits on the number of aspects within a substrate nor on the way in which aspects are arranged. In order to prevent disk surface analysis from providing useful information, aspects are able to migrate so that their physical footprints change completely over time. This migration can occur for mounted aspects without interrupting I/O and allows mounted aspects to exchange physical regions with aspects for which keys have not been entered. Steg does not require special offsets and aspects may reside within any region of a block device or file. Accordingly, Steg can also be used for steganographically hiding aspects within files.

DM-Steg consists of a Linux kernel module for mounting Steg aspects, as well as userland utilities for managing aspects and assisting the mounting process. The code is currently alpha, but works very nicely on my PC.

This document aims to outline the Steg specification and also give an overview of the workings of DM-Steg. Where this document fails to adequately define Steg, the DM-Steg source code should be authoritative.

2 What is deniable encryption?

Encryption of stored data is now a well established practice. Many tools are available to allow operating systems to automatically encrypt files or block devices. For Linux, dm-crypt provides many possibilities for block device encryption. This class of encryption may be characterised by the following properties:

- To obtain the plaintext, both the ciphertext and the secret key are required

- No effort is made to hide the existence of ciphertext

As such, this form of encryption works well when an attacker has an ability to obtain the ciphertext but has no ability to obtain the secret key. This is useful for eg:

- Hardware theft (eg. laptops, USB sticks)
- Remote attacks where the data is not currently mounted
- Hardware seizure by law-abiding government forces (in most jurisdictions)

If, however, the adversary is able to apply force in order to obtain the secret key (rubber-hose cryptanalysis), the overall security of the system is weakened considerably. Such adversaries may include:

- 'Private investigators' hired by criminal corporations (eg. BP, TEPCO)
- Government forces acting without regard to the law
- Courts ordering disclosure in jurisdictions with the required legislation

In the UK, part 3 of the RIP act came into force in 2008 and allows police to force suspects to decrypt any encrypted data that has been seized. Failure to provide plaintext may result in up to 5 years imprisonment. Put simply, forgetting your password can now be a criminal offense.

Deniable encryption presents a technical solution to the above problem. The core properties of deniable encryption are the inability to prove that:

- A suspected ciphertext is not simply random data
- A user has disclosed all of the keys for a known ciphertext

In the first case, it should be impossible to prove beyond all reasonable doubt that a suspected ciphertext is indeed a ciphertext. Potentially, this could be an effective legal defence but the user is still vulnerable to rubber-hose cryptanalysis. Rather than denying the existence of a ciphertext, the second method allows a given ciphertext to be decrypted with any number of keys. There is therefore no way of knowing if the user has granted access to all the plaintexts or merely a select subset. A clever user could divulge different keys at different stages of interrogation so as to give the illusion of complete disclosure.

3 Why Steg?

There are few software solutions offering deniable encryption at the present. The main contenders are:

Truecrypt: a popular disk encryption package for Windows, Linux, and Mac OS. Truecrypt provides high speeds, strong encryption, and some measure of deniability. On the downside, Truecrypt is distributed under a dubious (non-OSI-approved) license and its deniability support is rigid and limited.

Bestcrypt: another popular encryption package for Windows and Linux that allows multiple hidden volumes to reside within a single container. Unfortunately, Bestcrypt is neither free nor open source and requires all hidden volumes to be loaded before any are written to.

Rubberhose: a now defunct free and open source deniable encryption system for Linux 2.2. Rubberhose, like Bestcrypt, also required that all hidden volumes within a container be loaded before any are written to. This approach forces the user to enter sensitive passwords in the course of normal operation and risk accidental disclosure.

As all of the then-current offerings had serious shortcomings, the writing of Steg was justified.

4 The Steg specification

4.1 Basic concepts

When creating an aspect, Steg divides a given substrate into equal sized units called blocks, each aligned on a multiple of its own size. Blocks are at least 64 KB (although usually much larger) and are powers of 2 in size. Blocks may be scattered around a substrate such that their physical locations bare no resemblance to their logical locations. A block may be one of the following:

- The free block: not a block *per se*, but a blank region used for migration of an aspects' other blocks (shuffling).

- The header block: the first block accessed at load time and contains data required to initialise the aspect and decode the data blocks.
- Data blocks: these blocks contain the data for the block device exported to the user.

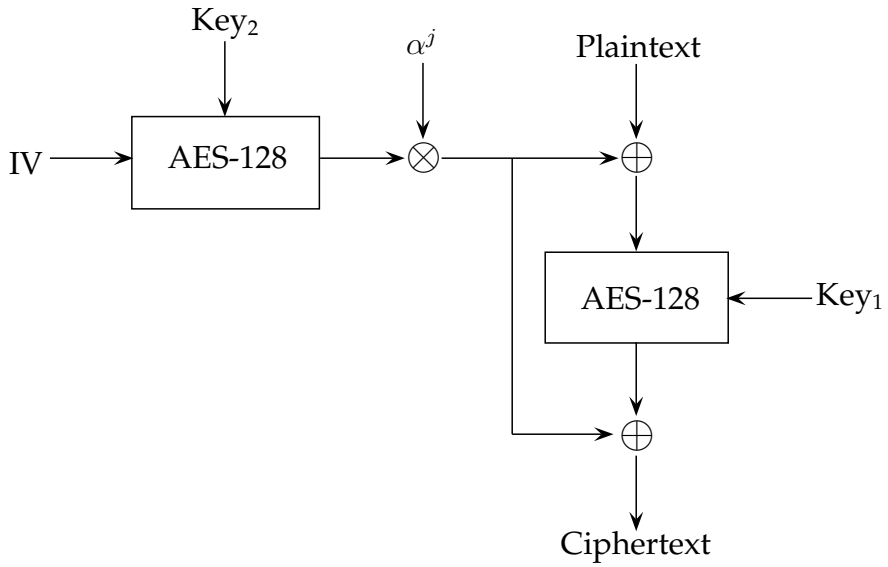
Blocks are subdivided into atoms - units of atomic access. Atoms are at least 512 bytes (one sector) but may be larger powers of 2. Like blocks, atoms appear in three different varieties:

- The header atom: 512 bytes, padded up to atom size (if necessary) with random data. Always at offset zero within the header block and required to load the rest of the aspect.
- Meta atoms: appearing only in the header block, meta atoms store meta data required for accessing an aspect.
- Data atoms: appearing only in data blocks, data atoms store the data of the block device exported to the user.

4.2 SXTS

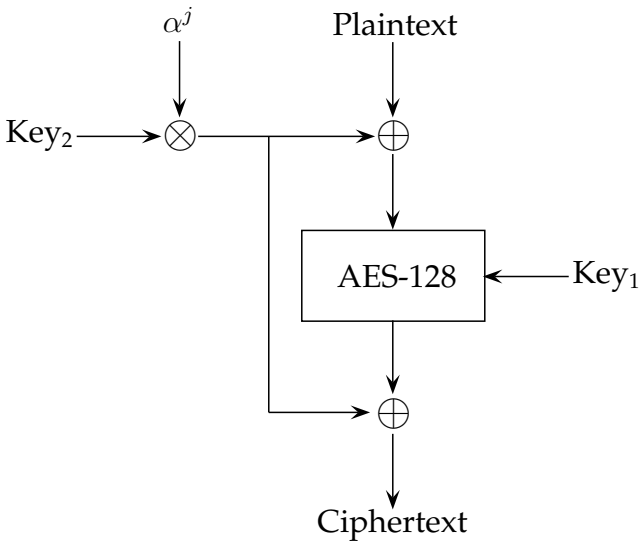
Traditionally, block device encryption software has used a single encryption key for the entire device and employed sector-specific initial vectors (IVs) to work around the otherwise-attendant security vulnerabilities. Steg takes a different approach and uses a secure hashing function, SHA-256, to generate unique keys for each atom in an aspect. This approach is more complex but offers better security.

The current gold standard in narrow-block data encryption is the combination of the AES block cipher with the XTS block cipher mode. XTS divides the given key into Key_1 and Key_2, where $Key = Key_1||Key_2$. Key_2 is used to encrypt a given initial vector (IV) to give an initial tweak value. The plaintext is then XORed with the tweak value, encrypted using Key_1, and re-XORed with the tweak value. The tweak value is then changed each round by a fixed field multiplication. Confusingly, XTS is described using the length of the full key. This means that AES-256-XTS actually uses 128-bit AES.



AES-256-XTS

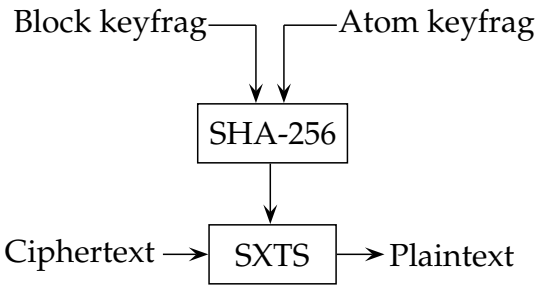
As Steg uses SHA-256 to generate unique keys for each atom, the initial tweak generation of XTS is superfluous. As such, Steg uses SXTS, a Shortened / Simplified version of XTS, in which Key_2 is used as the initial tweak. The rest of XTS's operation is unchanged. As AES-192 and AES-256 have significant weaknesses in their key schedules, Steg uses AES-128. The resultant combination of AES-128 and SXTS, known as AES-256-SXTS, is the sole cipher used throughout Steg. For systems without SXTS support, AES-256-SXTS may be emulated using AES-256-XTS (see DM-Steg source code).



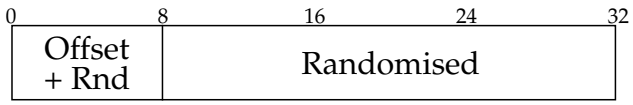
AES-256-SXTS

4.3 Accessing data within an aspect

Every data block in Steg has an associated 32-byte 'block keyfrag', a key unique to that block. Additionally, every atom's offset within its block corresponds to a 32-byte 'atom keyfrag', another unique key. An aspect therefore has as many block keyfrags as it has header and data blocks, and as many atom keyfrags as it has atoms in each block. Atoms may be thought of as residing on a 2D grid, with block number on one axis and atom number on the other. To derive the key for a particular data atom, the atom keyfrag is appended to the block keyfrag and the SHA-256 hash of the resultant 64 bytes taken. The resultant 256-bit key is used to decrypt the atom's ciphertext.



The physical location of each block is contained within the first 8 bytes of the block's keyfrag. This unit may be considered as a 64-bit unsigned integer and, like all integers within Steg, is little-endian. As each block is aligned on a multiple of its own size, the lower bits of this unit are not used for address calculation and serve purely for key generation. For example, for an aspect with 64 KB blocks, the least significant 16 bits of the block keyfrag's first 8 bytes would be read as zero, while the upper 48 bits would describe the block's physical offset within the substrate. To determine the physical offset of an atom, it's offset within its block is added to the physical address of its block.



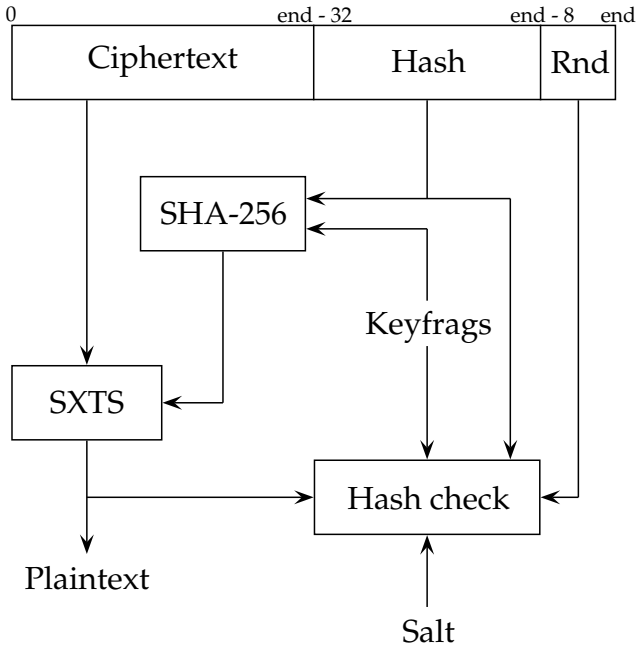
Steg's method of data atom key generation has the following advantages:

- Atom keys are unique, and unrelated to other atom keys.

- Compromise of an atom key does not compromise the parent keyfrags
- Moving a block changes the keys of all its atoms

Meta atoms reside within the header block and their physical addresses may be calculated in the same manner as for data atoms, using the header block keyfrag. Key calculation is more complex for meta atoms and uses extra data located in the last 32 bytes of the on-disk meta atom (the tail). First, one iteration of SHA-256 is performed on the 192-bit (24 byte) hash in the tail (extension implicitly performed as per the SHA-256 specification). A second iteration of SHA-256 is then performed on 64 bytes of data obtained by appending the atom keyfrag to the block keyfrag. The hash is then finalised and used as a key by which to decrypt the meta atom's ciphertext.

The plaintext is fed to a SHA-512 hash (extension implicitly performed as per the SHA-512 specification), followed by a single 128 byte block formed by concatenating a 56 byte salt (contained in the aspect's header atom), the 8-byte unit from the meta atom tail ('Rnd'), the block keyfrag, and the atom keyfrag. The first 24 bytes (192 bits) of the finalised hash is then compared to the hash stored in the meta atom tail. If the hashes match, the atom is deemed to be intact.



To encrypt a meta atom, 'Rnd' is randomised and a hash is calculated as described above. The hash, block keyfrag, and atom keyfrag are then used to calculate a key which is used to encrypt the plaintext. The ciphertext, hash, and 'Rnd' are then concatenated to form a full atom, which is written to the substrate at the appropriate location.

The meta atom encryption scheme has the following advantages:

- The whole of an atom's ciphertext changes each write, even if the plaintext does not
- Alteration of the ciphertext (malicious or otherwise) can be detected
- Even if an attacker knows the atom's plaintext, it's ciphertext and tail remain a mystery

4.4 Aspect loading

The header atom must be located and loaded before the rest of an aspect can be loaded. As a header atom can reside at any location, Steg prioritises locations in order of alignment, with the most aligned address being taken as zero. Amongst equally aligned address, lower addresses are prioritised. For a 1 MB substrate the order would be: 0x0, 0x80000, 0x40000, 0xc0000, 0x20000, 0x60000, 0xa0000, and so on. This is clearer when all sixteen of the potential block addresses within the 1 MB substrate are expressed in binary, highest priority first:

0000 0000 0000 0000 0000
1000 0000 0000 0000 0000
0100 0000 0000 0000 0000
1100 0000 0000 0000 0000
0010 0000 0000 0000 0000
0110 0000 0000 0000 0000
1010 0000 0000 0000 0000
1110 0000 0000 0000 0000
0001 0000 0000 0000 0000
0011 0000 0000 0000 0000
0101 0000 0000 0000 0000
0111 0000 0000 0000 0000
1001 0000 0000 0000 0000
1011 0000 0000 0000 0000
1101 0000 0000 0000 0000
1111 0000 0000 0000 0000

For each address, one sector (512 bytes) is read in. The SHA-256 sum of the latter 448 bytes is then taken and is XORed onto the first 32 bytes of the sector. The first 64 bytes of the sector is then put through the Bunny (see below), in calculate mode, which returns another 64 bytes of data. The SHA-256 sum of the returned 64 bytes is taken and used as a key with which to decrypt the latter 448 bytes of the sector. If the SHA-256 sum of the first 416 bytes of the plaintext matches the hash stored in the last 32 bytes of the plaintext, then the plaintext is a valid header. This encryption regime ensures:

- Any corruption of the first 512 bytes of the header atom renders the whole atom indifferentiable from pseudorandom data. Partial recovery of a deleted header is impossible.
- Any deliberate alterations to the ciphertext are detected
- To an attacker without the key, the header atom appears to be pseudorandom data

After the header has been successfully decrypted, the atom keyfrags are loaded. These are stored in meta atoms within the header block at an offset specified in the header atom. The atom keyfrags should be stored in the header block in such a way that one of the meta atoms is self-referential, ie. it contains the atom keyfrag required to load itself. This meta atom should also contain the atom keyfrags required to load the meta atoms before and after it. The atom keyfrag for the self-referential meta atom is stored in the header and is used to bootstrap the rest of the atom keyfrags.

After the atom keyfrags have been loaded, the block keyfrags are loaded. These are also stored in meta atoms within the header block at an offset specified in the header atom.

Finally, if shuffling is enabled, the journal atom is loaded and any incomplete shuffles rolled back (see below).

4.5 The Bunny

As Steg requires no special addresses and aspects may occupy any region of a substrate, decryption of many potential header atoms may have to be attempted. The header decryption algorithm was therefore designed with the following properties in mind:

- Long precomputation time
- Short per-atom decryption time
- Resistance to precomputed table attacks (rainbow tables)

The usual method for defeating rainbow tables is to combine the user-supplied passphrase with a salt before performing a time-intensive hashing. Unfortunately, Steg may require multiple ciphertexts to be evaluated, and the salting approach would result in an overall time proportional to the number of ciphertexts that must be evaluated. Key derivation functions such as PBKDF2 can be used but are also unsuitable for Steg for two reasons:

- Small working set lends itself to dedicated cracking hardware
- Key expansion rate slower than hard drive speeds makes precomputation advantageous

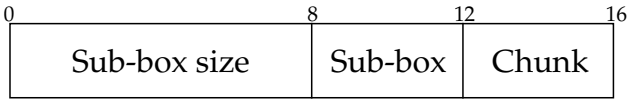
To meet the design requirements, a novel key expansion function was designed. This algorithm was designed to fit well with the capabilities of CPUs/GPUs and not lend itself well to faster execution on simplified hardware. The first step in the Steg key expansion algorithm is a multi-round hashing function, designed to protect the user's passphrase if the attacker were to discover the hash:

$$T_0 = \text{SHA-256}(\text{Salt} \,||\, \text{SHA-256}(\text{passphrase}))$$
$$T_n = \text{SHA-256}(T_{n-1} \,||\, \text{SHA-256}(\text{passphrase}))$$
$$\text{Final hash} = T_{8192}$$

The passphrase hash is then used to create a large amount of pseudorandom data. The total of the expanded data created by the Bunny function is known as the 'box'. The box is made up of one or many 'sub-boxes', each of which may be prepared independently from the others. Each sub-box in turn consists of at least 2^{16} entries of 64 bytes each, which are the working units for the Bunny state machine. Security levels are numbered from 0 upwards. Each level increment doubles both the number and size of the sub-boxes. Steg defines no upper limit and higher levels will be appropriate as faster hardware becomes available. When loading an aspect for which the Bunny level is not known, Steg may precalculate Bunny data for all levels between 0 and a default maximum, then test each potential header at every level. As each level increment increases the resource requirements by roughly 4 times, the cumulative memory and processing requirements for testing all levels below the maximum level should total no more than 50% of the requirements for testing at the maximum level only.

Level	Sub-boxes	Sub-box size	Box size	Box entries
0	1	4 MB	4 MB	2^{16}
1	2	8 MB	16 MB	2^{18}
2	4	16 MB	64 MB	2^{20}
3	8	32 MB	256 MB	2^{22}

Initially, each sub-box is filled with zeros. The sub-box is then divided into chunks of 4 KB and AES-256-SXTS used to encrypt each chunk. The key for each encryption operation is derived by XORing a 128-bit tweak value onto the upper and lower halves of the 256-bit passphrase hash. The first 8 bytes of the tweak is the size of the sub-box, in bytes. The next 4 bytes of the tweak is the sub-box number, starting from zero. The final 4 bytes of the tweak is the current chunk number, starting from zero within each sub-box.



$$\text{chunk key} = \text{passphrase hash} \oplus (\text{tweak} \,||\, \text{tweak})$$

After pseudorandomising the sub-box, the Bunny state machine is introduced and serves to further mix the data by 'hopping' around inside the sub-box. The Bunny has a 64-byte internal state (initially zero) and an 8-byte CRC. Starting at the first entry within its sub-box, the Bunny XORs the target entry onto its internal state. It then XORs the new internal state back onto the target entry. The CRC is then calculated by treating the state as containing eight 64-bit unsigned integers, multiplying them together, then multiplying the result by the hop, numbered from 1. The most significant bits of the CRC then describe the number of the next target entry.

```
number of hops = 2^(18 + level);
target = 0;
state = 0;
hop = 1;
for each hop {
        state = state ⊕ *target;
        *target = state ⊕ *target;
        crc = 1;
        for each u_int64_t in state {
                crc = crc × u_int64_t;
        }
        crc = crc × hop
        target = crc » (64 - sub-box bits);
}
```

To decrypt a header atom, the Bunny is operated in calculation mode. Calculation mode differs from precalculation mode in four ways. Firstly, instead of hopping around inside sub-boxes, the Bunny is given run of the full box. Secondly, the box contents are not modified during operation. Thirdly, the number of hops is much lower. Fourthly, the state is provided by the potential header atom rather than being set to zero. After performing the requisite number of hops, the current state is returned:

```
number of hops = 2^(14 + 2 × level);
target = 0;
state = <read from header>;
hop = 1;
for each hop {
        state = state ⊕ *target;
        crc = 1;
        for each u_int64_t in state {
                crc = crc × u_int64_t;
        }
        crc = crc × hop
        target = crc » (64 - box bits);
}
return state;
```

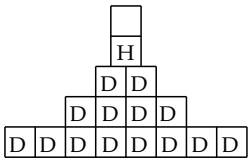
Operating the Bunny in calculation mode serves to check the integrity of the Bunny data and ensures that the whole dataset has either been loaded from

disk, or generated *in situ*. Either approach will slow down an attacker enough to prevent aspects with reasonably strong passwords from being discovered.

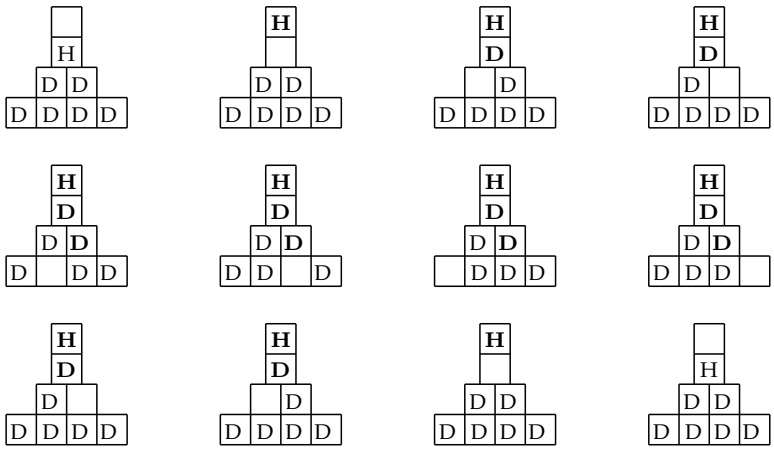
4.6 Shuffling

As stated previously, an aspect consists of a free block, a header block, and one or more data blocks. Shuffling is the process by which the contents of the header block or a data block are re-written onto the free block. When the process is complete, the source block is then marked as free and the process may continue. When a data block is being shuffled, the journal atom in the header block is updated before and after the meta atom containing the block keyfrag is updated. This ensures that power loss during a shuffle will not corrupt the aspect's meta data, or the user's data. When shuffling a header atom, no journalling is performed. Instead the 'sequence' entry in the header is set such that Steg may determine which header to use when loading an aspect.

As outlined above, header blocks should preferentially be stored at aligned offsets so that they may be located rapidly at load time. Additionally, transparent aspects (see below) require preservation of alignment within their data blocks. To allow shuffling to proceed without de-aligning any blocks, Steg internally organises block into a 'pyramid', according to their alignment. The free block, which should always be the most aligned, initially sits on the top layer of the pyramid. For an aspect with 64 KB blocks occupying the whole of a 1 MB substrate, the pyramid initially appears as follows:



The first shuffle occurs when the header block is promoted into the space marked by the free block. The second shuffle can then occur as a randomly-selected data block from the 3rd level is promoted into the new free area. The shuffling continues as the free block moves down the pyramid, completing the desired number of shuffles on each layer. When finished, it begins ascending back to the top of the pyramid. So that only blocks that have already been promoted are de-promoted during free block ascension, Steg keeps a list of promoted blocks in the journal atom. The process is shown here, with promoted blocks in bold:



When a block is shuffled, the bits in the new block keyfrag that are not used for addressing are randomised. Therefore, atoms in the new block have different keys to atoms in the old block and, over time, even if the contents of an aspect do not change, it's ciphertext does.

When creating a new header block, Steg should first write the atom keyfrags, block keyfrags and a clean journal. After they have been committed to disk, Steg may then write a new header atom, activating the new header block. The previous header will not be loaded by Steg, due to its less preferred sequence number, and will be overwritten the next time a shuffle occurs.

4.7 Transparent Aspects

Transparent aspects are aspects in which data atom encryption (but not header atom or meta atom encryption) is disabled, by use of the 'encryption' setting in the header atom. Transparent aspects should be created such that the alignment order amongst exported (user-visible) addresses mirrors that of their corresponding physical addresses. The envisaged use of a transparent aspects is as a layer that may be placed over the whole of a substrate so as to export another substrate of approximately the same size as the original. This allows encrypted aspects created within the transparent aspect to be rearranged on disk *via* the shuffling of the transparent aspect, even if they are not known to the system.

When creating a transparent aspect, its passphrase hash is randomised and does not need to be known to the user: it can instead be stored in memory by the

aspect management utility. When a user creates an aspect within the transparent aspect, the transparent aspect's passphrase hash will be stored in the new aspect's header atom. For loading an encrypted aspect within a transparent aspect, the process occurs as follows:

1. Load the encrypted aspect's header atom from the original substrate
2. Load the transparent aspect's header atom from the original substrate
3. Load the transparent aspect
4. Re-locate and load the encrypted aspect's header atom from within the transparent aspect
5. Load the encrypted aspect

Steg supports the use of transparent aspects within transparent aspects, and places no limits on the number of transparent aspects an encrypted aspect may ultimately reside within.

Transparent aspects do not compromise the deniability of Steg since, to an attacker without a key for an encrypted aspect within the transparent aspect, the transparent aspect's meta data would appear simply as random data.

4.8 Usage examples

Example usages of the Steg primitives and their advantages/disadvantages are shown here:

A single encrypted aspect spanning the whole of a substrate:

- Simplest use of Steg
- Strong encryption but very limited deniability

Multiple encrypted aspects spanning the whole of a substrate:

- Hidden aspects can be explained as free space
- Disk surface analysis may suggest otherwise

Multiple encrypted aspects within a transparent aspect:

- Hidden aspects can be explained as free space
- Probably very hard to prove otherwise

An encrypted aspect hidden at the end of an encrypted aspect's exported substrate:

- Nigh impossible to prove the hidden aspect
- Extra encryption/decryption would decrease performance
- Hidden aspect would be silently destroyed if the known aspect's filesystem was filled up

Although not part of the Steg specification, overlapping aspects are implicitly allowed and may be used to create an additional aspect within the data region of another aspect. In practice, this can allow an aspect to be hidden in the free space beyond the end of a filesystem. As the contents of a freshly created encrypted aspect appear random, the internal aspect would be fully deniable.

An encrypted aspect residing within the data blocks of another encrypted aspect, both inside a transparent aspect:

- Nigh impossible to prove the hidden aspect
- Shuffling must be disabled for the non-hidden aspect
- Hidden aspect would be silently destroyed if the known aspect's filesystem was filled up

Steg may also be used to hide data in a file. Steg requires multiple perfectly aligned regions of at least 64 KB each. If the file is stored inside a shuffled aspect, disk surface analysis should yield very little information.

Finally, although some uses of Steg may be fully deniable, users should be aware of the ever-present threat of the operating system leaking meta data and giving away the existence of a hidden filesystem.

5 DM-Steg

DM-Steg is the current implementation of the Steg specification. The userland tools are written portably and should work in any POSIX environment, while the kernel module is written specifically for Linux. All of DM-Steg should be considered alpha and should not yet be trusted with critical data. The current release of DM-Steg is dmsteg-0.9.tar.gz and its SHA-256 sum is:

0e166edbb104c316e0ae8ec9929a194563f0f78149b0862324ce742f2dbe86fb

5.1 Stegdisk

Stegdisk is the utility for aspect management. Stegdisk can erase substrates, create encrypted and transparent aspects, and import/export aspect contents from/to image files. Creation and editing of aspects-within-aspects is fully supported *via* the 'open' command.

Stegdisk allows substrates to be divided into 'regions' so that aspects can be created only within a selected part of a substrate, eg. for hiding aspects within files. Regions also facilitate the creation of overlapping aspects (see the 'steal' command).

For performance reasons, Stegdisk chooses a default block size based on the size of the substrate, usually in the order of a few megabytes. Also for performance reasons, the default atom size is 4 KB, equivalent to one page on most architectures. Requests sent to the kernel module appear to always consist of one or more aligned pages, and aspects with page-sized atoms exhibit a large performance increase over aspects with 512 byte atoms.

Stegdisk defaults to creation of encrypted aspects with bunny level 1 and transparent aspects with Bunny level 0. When loading aspects, Stegdisk will by default search for aspects at levels 0 and 1.

5.2 Kernel components

The Linux kernel components consist of the DM-Steg device mapper module, the SXTS module, and miscellaneous support files.

The DM-Steg module is written for the device mapper infrastructure and provides a layer that can be placed over a substrate so as to export an aspect. The kernel module works independently with each aspect and knows nothing about the relationships between aspects. Shuffling is triggered by device mapper messages sent from userland.

5.3 The Steg Daemon

The Steg daemon, stegd, keeps a record of live aspects and sends out periodic shuffle messages. stegd also invokes the Bunny to calculate new keys when the header block must be moved. To stop sensitive data being leaked, mlockall() is used. Also, like all DM-Steg userland tools, stegd zeros memory before freeing it. stegd listens on /var/run/stegd for incoming connections from stegsetup and stegctl (see below).

As DM-Steg is currently alpha, automatic shuffling is disabled by default. To change this, use stegctl.

5.4 stegctl

stegctl sends messages to stegd *via* the /var/run/stegd socket and prints the reply. stegctl is used by stegmount and stegumount and can also be used for turning on and off automatic shuffling ('stegctl rate 0/1'), and for manually triggering individual shuffles ('stegctl shuffle'). See stegd.c for a full list of commands.

5.5 Mounting tools

To instruct the kernel module to setup an aspect, the stegsetup utility is used. It works as follows:

1. Prompt the user for a passphrase
2. Perform the Bunny precomputation
3. Scan the substrate for headers
4. Test-load the aspect and, if necessary, clean its journal
5. Ask stegd for an unused /dev/mapper/stegX device

6. Instruct the device-mapper kernel components to set up the aspect
7. If successful, register the aspect with stegd

stegsetup can handle nested aspects and will set up as many /dev/mapper/stegX devices as required.

stegmount is a wrapper for stegsetup that tries to mount the innermost aspect's filesystem at a provided mount point.

stegumount unmounts a filesystem, unloads an aspect, and removes any unused supporting aspects.

6 Benchmarks

All benchmarks were taken on my system with 4 GB RAM, a 1.5 TB Spinpoint F2 EcoGreen SATA HDD, running an ancient Gentoo install with:

Linux shakti 3.2.0-rc3SHAKTI #1 SMP Thu Nov 24 17:02:53 GMT 2011 x86_64 Intel(R) Core(TM)2 CPU 4400 @ 2.00GHz GenuineIntel GNU/Linux

To test whole-device read performance, 'time cp <device> /dev/null' was executed. To test the corresponding write performance, 'cp /dev/zero <device> ; sync' was timed. The substrates used for benchmarking were /dev/sda1, an aligned 32 GB partition, and /dev/ram0, a 2 GB ramdisk with 4 KB blocks. Whole-device reads and writes were performed on the substrates themselves, on encrypted aspects within the substrates, and on encrypted aspects within transparent aspects within the substrates. All aspects were of the maximum possible size and used 4 KB atoms. The numbers reported are in MB/s:

Device	Read		Write	
/dev/sda1	110		111	
Block size:	4 MB	32 MB	4 MB	32 MB
sda1(enc)	88	106	73	81
sda1(trans(enc))	88	107	73	79

Device	Read		Write	
/dev/ram0	691		578	
Block size:	1 MB	32 MB	1 MB	32 MB
ram0(enc)	121	121	148	148
ram0(trans(enc))	129	127	141	141

Kernel compiles were performed to test real-world performance. A ReiserFS filesystem was created on /dev/sda1, the kernel source tree was copied across, caches were dropped, and a 'time make' was performed. The same procedure was then performed inside an encrypted aspect using 4 MB blocks and taking up the whole of /dev/sda1, and the finally inside an encrypted aspect inside a transparent aspect on /dev/sda1, both of maximum size and both using 4 MB blocks. The three compiles took 930, 938, and 933 seconds, respectively, giving a maximum performance penalty of less than 1%.

7 Credits

Steg and DM-Steg were developed by Leopold Samulis (anagon@gmail.com) from 2010-2011.

With infinite gratitude to Mj, for teaching me crypto.

8 Future Work

I have taken Steg/DM-Steg as far as I want and any further improvements will be left to the OSS/free software community at large. Aside from bug fixing there are a handful of things that could really use doing, such as:

- Proper idle detect for stegd
- Optimizing the kernel code
- Wear levelling for the journal atom
- Wide-block encryption with authentication for data atoms
- GRUB support!

If you like what you see and are interested in contributing or taking over leadership of this project, please contact me at anagon@gmail.com.

9 License

The Steg specification should be considered free and unencumbered. This document and it's source .tex file are hereby released into the public domain.

DM-Steg is hereby released in its entirety into the public domain. This means you may do with DM-Steg whatever you please, wherever you please, whenever you please, inasmuch as you please, for the rest of eternity. Having said that, I would kindly ask that you:

1. Release all modified and derived works into the public domain
2. Do not deviate from the specification without very good reason
3. Give credit where credit is due

Also, please see the below boilerplate:

THIS SOFTWARE IS PROVIDED BY LEOPOLD SAMULIS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LEOPOLD SAMULIS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. LEOPOLD SAMULIS AND CONTRIBUTORS SHALL ALSO NOT BE LIABLE IF USE, POSSESSION, OR TRANSMISSION OF THIS SOFTWARE GETS YOU IN TROUBLE WITH WHICHEVER LEGAL SYSTEM YOU LIVE UNDER.