The Dissident File System

Aneesh Neelam
Univ. of California, Santa Cruz
aneelam@ucsc.edu

Darrell D. E. Long
Univ. of California, Santa Cruz
darrell@ucsc.edu

May 19, 2016

Abstract

Dissidents under authoritative governments may not have the equivalent of the United States' Fifth Amendment protections. Therefore cannot just encrypt their data without being subjected to rubber hose attacks. Also, traditional steganography cannot be used under such circumstances, as it will be broken eventually by the State, given enough time and effort. Even in chaotic systems, there will be a pattern of limited predictability. The dissident file system combines aspects of steganography and encryption to protect data from untrusted parties, and attempts to provide the dissident with a means of plausible deniability.

1 Introduction and Assumptions

In this paper, we shall talk about the design and implementation details of the dissident file system. Some explanations on why certain algorithms, tools, and design patterns were chosen along with their tradeoffs are also described in detail in this paper. However, this design and implementation shall be improved upon later as we have detailed in Section 6.

The dissident file system does not protect the dissident from being discovered via other sources, such as online or physical surveillance, etc. It only serves to protect the data on the storage device.

2 Design

The dissident file system uses harmless, "innocent" files to "encrypt" sensitive files. It is a layer on top the existing file system. The objective is to ensure that these sensitive files cannot be made sense of without making use of the dissident file system. The dissident file system must be a layer on top of the existing "normal" one, as the very absence of any normally readable files on the disk is a red flag.

Innocent files are those which are completely legal in the specific jurisdiction and are not suspicious for the dissident to have on his or her system. Since, these files are generally specific and customized for the dissident, they must be able to provide them for use in the dissident file system.

However, encryption is not performed in the usual sense in the dissident file system. The sensitive file data when written, shall be XORed with an innocent file and the result is written to the underlying file system. And when reading these files via the dissident file system, the underlying file is then XORed with the corresponding innocent file to obtain the actual data.

We know that if r is random, then $q = r \bigoplus d$ is also random, regardless of what d is. However, the "innocent" file cannot be made up of random data. otherwise it cannot be termed innocent anymore.

Suppose, the sensitive data that is to be written via the dissident file system is a byte array d. The innocent file that must be XORed with d can be a. Now, assuming that the innocent file is much larger than the sensitive file, the byte array d will be XORed with an offset, i, and onward, until the size of d. This offset i is randomly generated when writing a new file. Otherwise it is retrieved from a store. The pseudo-random number generator must be cryptographically secure; else the offsets can be reverse engineered by analyzing the dissident file system in

execution, or even compromise the security of other instances of the dissident file system on other machines if the random numbers generated are predictable.

The file inode numbers and the respective offsets must be written to a file system store, as a file on the underlying file system itself. A key-value pair of inodes and specific offsets, and some other information such as the file path to the respective innocent file and other file system stores may also be stored.

This file system store is also a file that understandably needs to be protected in the same manner. However, the corresponding offset for this must be determined using a deterministic method. The offset can be chosen based on the cryptographic hash of a pass phrase entered when mounting the dissident file system.

3 Implementation

The dissident file system is built using the FUSE API for POSIX systems. This enables compatibility across the different Unix-based and Unix-like systems. But, some aspects of the design and implementation may depend on how the POSIX interface is implemented in that operating system. Hence, we shall assume that this file system will be used on FreeBSD, Linux, and Mac OS X systems.

For the file system to work, we need a mechanism to generate random numbers for offsets. We also need a secure hashing algorithm for calculating the offset from the passphrase for the root file system store. And we need a file system store that provides durability and consistency while being able to handle concurrent file system operations.

3.1 Mounting

To mount the file system, the parameters required are: the source directory, a target directory/mount point, a passphrase and the file path to the root file system store and the file path to the root innocent file. If the root innocent file is not specified, the dissident file system assumes this to be the first time use, in which case the directory with the innocent files shall also be requested. But, to prevent mistakes, the root file system store file is not overwritten unless explicitly desired by the dissident.

3.2 Passphrase

The offset for the file system store is calculated using SHA-256, one of the algorithms in the SHA-2 family of cryptographic hashing algorithms. Running SHA-256 on the passphrase results in a 32 byte hash. Taking the most significant byte as the offset, we can then read the root store file. Every subdirectory shall also have its own innocent file and a store, that can be read from the root file system store. These sub-innocent files and sub-stores shall apply to their respective subdirectory.

3.3 First Time Use

The file system assumes that the file system is being used for the first time if the root innocent file is not present, and the user chooses to either ignore and overwrite an existing root file system store or provides a path for a new file system store to be created. A directory with the dissident's innocent files must be specified. The path to this directory is also stored in the newly created or truncated file system store.

3.4 Random Number Generator

For generating cryptographically secure random numbers, only hardware-based or at least hardware-backed random number generators must be used. Assuming the dissident will not be able to carry specialized hardware random number generators, we can only rely on hardware-backed ones.

We use the /dev/random device file, which uses environmental noise for its entropy pool to obtain the random seeds for the pseudo-random number generator. Implementations of /dev/random by all three systems, are designed to be cryptographically secure and even long lasting keys like SSH, GPG and SSL keys are generated from such hardware-backed entropy systems. /dev/random blocks until a sufficient entropy pool is present to generate a secure

random number, a timeout could be present or the user could be instructed to perform tasks on the computer to generate entropy.

Since, using /dev/random depletes the entropy pool and each read takes in a fixed stream of bytes, we shall store this sequence and read from /dev/random only if the stored sequence of random numbers are used up. The size of the offset take from the sequence each time will be based on the size of the innocent file. If the innocent file is not larger than the offset taken from the stream of random bytes, sequences of random numbers are only being wasted, depleting the entropy pool unnecessarily.

We can assume that the dissident file system shall be used on system which has user interaction, and not a server which does not have these sources of entropy. However, some servers also require cryptographically secure random numbers for securing data on the network. Hence, such servers are configured to obtain entropy for random number generation from other sources on the internet.

/dev/random Implementations:

- On Linux systems, the /dev/random device takes environmental noise from device drivers into an entropy pool. From this pool cryptographically secure random numbers can be generated. On Linux, however, /dev/urandom does not block if a sufficient entropy pool is not present. Hence, /dev/urandom on Linux is not recommended for cryptographic algorithms.
- On Mac OS X, the /dev/random device uses the Yarrow algorithm devised by John Kelsey, Bruce Schneier and Niels Ferguson, and implemented by Apple. It uses various sources of entropy to seed its random number generator.
- On FreeBSD, the /dev/random device uses the Fortuna pseudo random number generator by Bruce Schneier and Niels Ferguson, and implemented by the FreeBSD developers. It also uses various sources of entropy to seed its random number generator, and is similar to Yarrow.

3.5 File System Store

BerkeleyDB is an embedded key-value database library that can be incorporated into applications. It supports concurrent requests, which are necessary for a file system. It is supported on FreeBSD, Linux, and Mac OS X.

Since the inode numbers do not change unless the file is moved, it can be used as a key for the file system store. The offsets must be stored with the respective inode numbers in the dissident file system store. For a randomly generated offset i, and a file size s, the corresponding innocent file needed ranges from i to $i+s$. If $i+s$ becomes greater than the size of the innocent file itself, the modulus operator can be used to go back to the start of the file. This provides an apparent endless continuity to the innocent file.

When appending or updating data in-place to an existing file, one must calculate the starting offset by taking the existing offset and adding the specified offset or the original file size to obtain the offset of the innocent file to XOR with.

4 Trade offs

Here we talk about some of the problems that this design and implementation can have. We hope to perfect the design as we go along.

The most significant byte of the cryptographic hash of the passphrase is chosen to be the offset of the root file system store. However, this can be a parameter given when mounting if necessary. 1 byte would mean that the innocent files must be at least 4 Gigabytes to be fully effective, but it is not an absolute requirement.

Deliberate manipulation of the sensitive files or even the innocent files from the underlying normal file system can destroy the data. The underlying file system does not do so as part of its normal execution, the user themselves or a program must deliberately be instructed to manipulate the data from the underlying file system for the sensitive files to be destroyed. The file names (and paths) are still unique in this case, and hence, the underlying normal file system would not overwrite these files unless explicitly performed.

We are not XORing random data at all; we are only choosing non-random innocent data at random. This is a tradeoff as it may be more statistically insecure than using actual random data. However, we assume that innocent files are easier to pass off than random files. Also, if the innocent files themselves are compressed files, the entropy of the innocent file itself is higher, making the innocent data itself more "random-like".

There are still XORed files all over the file system; these represent the actual sensitive data. However, since these files are generally smaller in number and smaller in size when compared to the "innocent" files used, the dissident may be able to pass them off as file system corruption. As part of future work, we can look at storing these files on a separate partition that is seemingly unallocated when scanning the disk, or on the free space of existing partitions. But specialized storage hardware that may attempt to hide this information may not be feasible as the very presence of such a device raises a red flag.

5 Analysis

In this section we talk about the performance and also some attack vectors.

Since, XORing data from multiple different sources requires more computation than simply reading one source to read a file, it is expected to be much slower than the underlying normal file system. However, since that is not the goal this overhead can be termed acceptable.

The adversary is assumed to be a government which is highly motivated and capable with almost unlimited resources. To reverse engineer the offsets, one must be able to break the randomness of the machine, and the SHA-256 algorithm. Since, these algorithms are known to be cryptographically secure for now as breaking them is computationally unfeasible for the foreseeable future. Therefore, we can use them to generate random numbers for offsets and to hash the passphrase.

It is possible to break the randomness of the machine by compromising the entropy sources. However, that only compromises the random numbers generated since then, not the ones already generated before.

6 Future Work

Since storing the sensitive data, even in XORed form on the disk as files are still visible on the underlying file system. This can be highly suspicious and opens up the dissident to rubber hose attacks. We hope to be able to store this data in the free blocks or space of the underlying file system or disk. That would require special error correcting codes and recovery methods as the free space can be overwritten by the underlying file system when writing normally, destroying the sensitive data.

References