DiFUSE - A 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

June 6, 2016

Abstract

Dissidents under totalitarian governments may not have the equivalent of the United States' Fifth Amendment protections. Therefore, one 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. We present DiFUSE, a dissident file system that 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

Dissidents use many tools to protect their digital data, online identity, etc. from a totalitarian government, and each of these tools has advantages and disadvantages. Depending on the laws of the State and the legal precedents established, the dissident may be compelled to reveal incriminating data to the State [7]. Failure to do so can lead to severe fines and incarceration. Therefore, it is not enough for the dissident to use the tool to conceal data or identity, but also conceal the use of such a tool from the State.

Encryption by itself does not hide the data. If a third party analyzes the data blocks, they may not be able to make sense of them, but they can conclude that it has been encrypted. They can then ask for the key. In countries such as the United States, you cannot be legally compelled to reveal anything that incriminates you [5]. However, not all States have such legal protections. Unfortunately, there are some which even torture for such information. An encrypted file system cannot be read without the decryption key(s), but such a State can compel the dissident to decrypt it or share the decryption key(s) [1, 7]. Steganography by itself is also not a feasible option. By analyzing the size of the data and the content, anomalies will be found and the use of steganography can be detected [17]. The State can eventually retrieve the dissidents data from the disk drives.

DiFUSE, a dissident file system, aims to provide plausible deniability for the dissident if their disk drives are ever subjected to inspection. In this paper, we shall talk about the design and implementation details of DiFUSE. Some explanations on why certain algorithms, tools, and design patterns were chosen along with their trade-offs are also described in this paper.

In the next section, we shall see the approaches that others have taken to solve this problem of safeguarding data while maintaining plausible deniability. In Section 3 and 4, we shall describe how DiFUSE is designed and the current implementation. In Section 5, we analyze its limitations and evaluate its performance. We then discuss future work in Section 6.

2 Related Work

A lot of work has been done to protect data stored on disks from third parties. Various encryption algorithms have been devised and implemented in a variety of languages and for many platforms. There are two primary kinds of encryption for data stored on disks: Full Disk Encryption and File-level Encryption.

Full Disk encryption encrypts the entire disk, including the free space [8, 14, 19]. There are ways to defeat this mechanism, however. It is possible to use a keylogger to capture the key when the user is entering it. Another method is called a cold-boot attack, where the key may be stored in RAM for comparison and since DRAM and SRAM are still readable for a few seconds after the system is shutdown, the contents of the RAM may be dumped and analyzed [12]. Various mechanisms to speed up disk encryption have been developed. Newer CPU architectures

have special AES instructions and specialized hardware for AES encryption [15].

File-based encryption, on the other hand, encrypts files individually in the file system layer and stores them. When reading the data back, the file system seamlessly decrypts the data. IBM developed Encrypted File System (EFS) for the AIX system, which employs this method to safeguard files stored on disk [13].

Both disk and filesystem level encryption suffer from the same problem: The existence of the data protected using encryption and the use of encryption mechanisms cannot be denied [7], and they are frequently subjected to rubber hose attacks [3].

Steganography has also been employed to safeguard data on disk. The Steganographic File System was developed to give users an element of plausible deniability that encryption alone cannot provide. Users cannot be compelled to divulge the key if the adversary is not aware that the data even exists. The user can confidently deny the existence of the data in such a case [1]. But, if data hidden using steganography can be found with some level of effort and time. This data is generally not encrypted as that can break steganography, and subsequently does not offer plausible deniability.

An encryption tool called Truecrypt was developed that also had a plausible deniability feature. The disk is split into two volumes. One volume contains the actual data; the other contains dummy data that the user is willing to sacrifice. Two keys are used to decrypt the disk. One key only decrypts one of the volumes. Hence, the user can give away the key corresponding to the volume with dummy data, if compelled and the sensitive data in the other volume is still safe [7]. But, the disk can be carefully analyzed and the size of the volume would not add up to the size of the disk, in which case the adversary can compel the user to disclose the actual key [6].

3 Design

DiFUSE has been implemented using the FUSE API, which is supported by many POSIX systems, including OS X, Linux, and FreeBSD [10]. The dissident file system is a layer on top of the existing "normal" one. This is because a fully fledged custom file system or the presence of a custom kernel may be suspicious, depending on how the totalitarian government perceives the dissident.

The dissident file system uses harmless, "innocent" files to "encrypt" sensitive files. Innocent files are those which are completely legal in the specific jurisdiction, are not suspicious for the dissident to have on his or her system and are also specific to the dissident. 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. Random unreadable data is also suspicious to have.

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 DiFUSE in execution, or even compromise the security of other instances of DiFUSE 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. 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 is chosen based on the cryptographic hash of a pass phrase entered when mounting the dissident file system.

4 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 file system store file. And we need a file system store that provides durability and consistency while being able to handle concurrent file system operations.

4.1 Passphrase

The offset for the file system store is calculated using SHA-512, one of the algorithms in the SHA-2 family of cryptographic hashing algorithms. Running SHA-256 on the passphrase results in a 64-byte hash. Taking the most significant byte as the offset, we can then read the store file.

4.2 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.

We can assume that the dissident file system shall be used on a 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 [21].

- 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 [16].

- On FreeBSD, the /dev/random device uses the Fortuna pseudo random number generator devised by Niels Ferguson and Bruce Schneier, and implemented by the FreeBSD developers. It also uses various sources of entropy to seed its random number generator and is similar to Yarrow [9].

4.3 File System Store

BerkeleyDB is an embedded key-value database library that can be incorporated into applications [18]. 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 [20]. 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.

5 Evaluation

We evaluate the performance, and also analyze the possible threats and attack vectors for the current implementation. We also some describe some of the trade-offs in the design.

5.1 Threats and Trade offs

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-512 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. Brute-forcing the offsets can also be attempted. However, the feasibility of this task depends on the size of the innocent file and the number of sensitive files.

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

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" [2].

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. However, we understand that this is a major concern as rubber hose attacks become possible [3] and as part of future work, this can be addressed.

5.2 Performance

DiFUSE's performance was evaluated and compared to the native file system present. All tests were performed on commodity hardware, in this case, a 15" Macbook Pro (mid-2015). The native file system is the default HFS+, and Mac OS X's FileVault disk encryption is enabled.

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

The following tools were used for benchmarking:

- dd & sync
- bonnie++

The read performance of DiFUSE is slightly above the native file system. We believe this anomaly occurs because the read operation is performed on the same file every time, and hence the same chunk of the innocent file is taken and XORed with the sensitive file. This chunk can be cached in memory by the operating system, thereby improving performance.

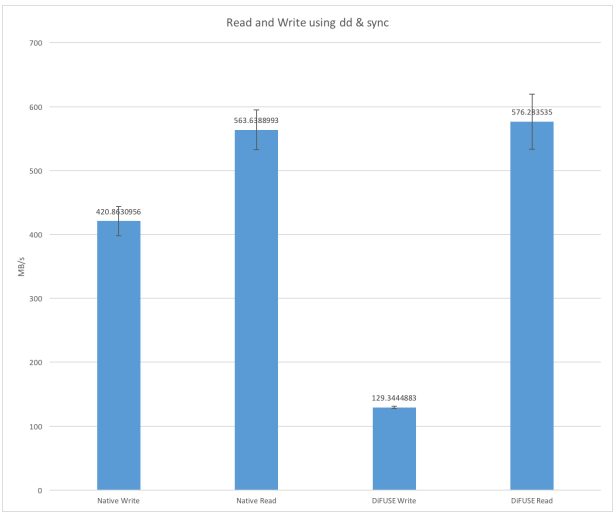

Figure 1: 18 runs of dd & sync of 1000 blocks of 100k size each.

The Bonnie++ 1.9.7 benchmark test large file IO and also the creation and deletion of small files [4]. Bonnie++ on DIFUSE has been used for the following benchmark.

	Sequential Output						Sequential Input				Random Seeks	
Size	Per Char		Block		Rewrite		Per Char		Block			
	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU
6G	15	25	81171	20	60418	24	1608	98	178821	23	5458	156
Latency	727ms		31968us		45134us		19478us		25746us		4546us	

Figure 2: Bonnie++ on DiFUSE - 6 GB large file: Throughput, CPU usage and Latency

	Sequential Create						Random Create					
Num Files	Create		Read		Delete		Create		Read		Delete	
	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU
16	849	43	22671	34	3432	31	798	42	31685	39	3704	31
Latency	23533us		818us		24095us		9130us		728us		12622us	

Figure 3: Bonnie++ on DiFUSE - 16 small files: Throughput, CPU usage and Latency

6 Future Work

Since, the presence of sensitive XORed data as normally unreadable files on the native file system is suspicious, a new method to store this data on the disk must be devised.

One such method could be storing in the free blocks of the underlying file system. Free space on a file system normally contains previously deleted data. When a disk is wiped using a secure method of erasing data, these free

blocks are indistinguishable from random data. Hence, the sensitive files could be stored here. However, the native file system may overwrite this data as it considers it free space. Therefore, redundancy and error correcting mechanisms could be used for this sensitive data. Redundancy and error correcting codes enable the dissident file system to reconstruct the sensitive data blocks if the data is ever overwritten.

Another method could be to mark the blocks where the sensitive data is stored as bad blocks [11]. Bad blocks are considered to be permanently damaged, and the native file system avoids them when writing to disk. Hence, "fake" bad blocks could be marked, and sensitive data could be stored on them. Actual bad blocks on the disk can be avoided normally.

However, both approaches may require access to the underlying file system's data structures from the dissident file system. We are looking to see if that is technically feasible without modifying the underlying file system, and also if it can be done in a file system agnostic manner.

7 Conclusion

DiFUSE is a work in progress towards a true dissident file system, where sensitive information could be stored on the disk, encrypted and also hidden from any external parties. A file system for dissidents allows them to resist totalitarian regimes with greater data security and physical safety from rubber hose attacks.

Acknowledgement

We would like to thank Ethan Miller for his suggestions and input. We would also like to thank the students of the Storage Systems Research Center for their feedback on the design of the Dissident File System.

References

[1] R. Anderson, R. Needham, and A. Shamir. *Information Hiding: Second International Workshop, IH'98 Portland, Oregon, USA, April 14–17, 1998 Proceedings*, chapter The Steganographic File System, pages 73–82. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[2] G. E. Blelloch. Introduction to data compression. *Computer Science Department, Carnegie Mellon University*, 2001.

[3] H. Bojinov, D. Sanchez, P. Reber, D. Boneh, and P. Lincoln. Neuroscience meets cryptography: Designing crypto primitives secure against rubber hose attacks. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 129–141, Bellevue, WA, 2012. USENIX.

[4] R. Coker. Bonnie++.

[5] Cornell. An overview of the fifth amendment.

[6] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the tattling os and applications. In *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, pages 7:1–7:7, Berkeley, CA, USA, 2008. USENIX Association.

[7] Defuse.ca. defuse.ca/truecrypt-plausible-deniability-useless-by-game-theory, 2013.

[8] N. Ferguson. Aes-cbc + elephant diffuser: A disk encryption algorithm for windows vista, 2006.

[9] N. Ferguson and B. Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[10] FUSE. github.com/libfuse/libfuse, 2016.

[11] P. Garvin and H. Stanard. Method and system for managing bad areas in flash memory, July 10 2001. US Patent 6,260,156.

[12] J. Götzfried and T. Müller. Analysing android's full disk encryption feature. *JoWUA*, 5(1):84–100, 2014.

[13] IBM. Understanding encrypted file system (efs), 2008.

[14] Intel. Whitepaper: Disk encryption in the enterprise, 2010.

[15] Intel. Whitepaper on intel advanced encryption standard (aes) new instructions set, 2010.

[16] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Selected Areas in Cryptography*, pages 13–33. Springer, 1999.

[17] G. C. Kessler. An overview of steganography for the computer forensics examiner, 2004.

[18] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[19] Symantec. Whitepaper: How whole disk encryption works, 2010.

[20] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[21] T. Ts'o. random. c - linux kernel random number generator.