Improving the Security of Static Data on Cloud Storage Platforms

Zachary Daniel Leaf
School of Electronic Engineering and Computer Science
Queen Mary University of London
London, UK
zdleaf@zinc.london

***Abstract*—There are significant security risks associated with storing static data on cloud storage platforms. This paper examines these risks, presenting practical solutions to mitigate these. In particular, the focus is on the application of sound cryptographic principles to ensure the confidentiality, authenticity and integrity of stored data. This will be of use to software developers or organisations who intend to use public cloud storage to store data.**

The shortcomings of existing solutions are also explored and improved upon.

The result of this paper is the development of an open source tool for securely syncing files between a Linux machine and cloud storage platforms, with a focus on ensuring the confidentiality, authenticity and integrity of data. The full code is available at https://github.com/zdleaf/enclone

***Index Terms*—Cloud computing; Data privacy; Encryption; Cryptography; Secure storage (search for more IEEE keywords)**

I. Introduction

As Infrastructure as a Service (IaaS) and cloud storage has now become common place, this presents significant security risks associated with storing data on servers outside of the user's control. The concern over data security and privacy has been said to be the biggest hurdle facing cloud adoption [2, 14].

A main benefit of cloud storage is that it is abstracted from the physical implementation of disks and server administration. Users can conveniently and instantly access large amounts of storage space, with a great deal of flexibility and reliability, without having to be concerned with the underlying infrastructure, maintenance and admin costs [2,3]. However, the result of this is that user does not know where their data is stored, or what processes are running on that server [1]. Users are now reliant on third parties to protect and secure their data against security breaches, from both internal "insider" and external attacks [references], and both hardware and software vulnerabilities that potentially expose confidential data [references]. Worse still, the virtualisation software used by cloud providers exposes completely new and additional vulnerabilities on top of existing threats [5].

It is also often unclear how the data stored on these services is used or shared by both the provider and their third-party affiliates. This is particularly true for personal cloud storage solutions such as Dropbox and Google Drive. For example, the Dropbox Terms of Service clearly and explicitly states it gives both Dropbox and their affiliates and "trusted" third parties full access to store and scan uploaded data [Dropbox TOS citation]. Further, the physical or operational resources may be themselves outsourced to third parties in a way that is not always transparent.

The overwhelming majority of solutions are, by default, insecure out of the box and vulnerable to a number of risks without additional hardening. It has therefore become essential to understand these risks and design and implement a strategy to ensure the confidentiality, authenticity and integrity of data stored on cloud storage platforms. A key point here is that trust is not synonymous with security [2]. Any security that is solely reliant on social means, i.e. "legal, physical or access control mechanisms" [14] such as promises made in Terms of Services or relying on the trustworthiness of a provider's employees to not access confidential data is insufficient [2, 14].

The purpose of this paper therefore, is to evaluate and implement technical and practical cryptographic methods to guarantee as far as possible the security of data stored on these services, while retaining the benefits of public cloud storage systems. The findings are implemented in a Linux utility/daemon to sync/backup files and directories from a local file system to public cloud platforms, in a cryptographically secure way.

II. Potential Risks

A. External attacks

The number of potential external attacks are large. Due to the reliance of cloud providers on virtualisation to maximise server utilisation, data is stored in shared environments. Attacks on the hypervisor, the software that handles these shared environments, such as Virtual Machine (VM) hopping and escape attacks [5, 12, 57] allow attackers to escape the VM into the host system or to other VMs on the same hardware. The VENOM vulnerability (CVE-2015-3456) is a recent and infamous example of how it is possible to "escape from the confines of an affected virtual machine (VM) guest and potentially obtain code-execution access to the host" [57]. In this way traditional security vulnerabilities can be used to exploit and gain access to poorly configured or vulnerable VM

instances to gain access to other VMs on the system and the wider network.

Side channel attacks, such as exploiting the shared physical memory on systems has allowed attackers to extract private keys and other information from VMs running on the same hardware [12, 23]. The most infamous of such side channel attacks are the MELTDOWN [60] and SPECTRE [61] vulnerabilities that allowed attackers to extract information from other processes running on the same CPU. This is a critical security issue in a multi-tenant cloud server containing confidential information.

Access to cloud platforms typically includes a web based interface as well as an API which provides further vectors for attack [3]. In almost all cases full access to stored files is possible through such web interfaces. Web based vulnerabilities are numerous, ranging from SQL injection, Cross Site Scripting (XSS), Cross Site Request Forgery (CSRF) and exploitation of session management, auth tokens and authentication systems. There is also, of course, the possibility for misconfiguration of cloud services by the user or compromise of user credentials through social engineering or phishing. All of these are additional vectors exposed by such web interfaces and APIs.

B. Internal "insider" attacks

In the Frequently Asked Questions (FAQs) for Amazon Web Services' (AWS) cloud storage platform - S3, it states that Amazon does not "access your data for any purpose outside of the Amazon S3 offering, except when required to do so by law" [citation]. However there remains the possibility that any insider working for AWS with sufficient permissions can access your raw data as they have access to the physical memory.

Similarly, while Dropbox's marketing material states that data is fully encrypted, the encryption keys are stored by Dropbox, giving them full access to your data at any time. Dropbox clearly states in [56] that a number of employees have full access to user data. This opens the possibility for a "malcious insider" attack – that is someone working within the cloud storage provider is be able to access to that data [7,9,11].

C. Network attacks

As data is transmitted across open computer networks to public clouds, there is the risk of Man in the Middle (MITM) and network sniffing attacks. SSL/TLS offers protection on data transmitted across most open networks, however the transmission on the internal networks of cloud storage providers is unknown. Further there have been previously been serious vulnerabilities in the implementation of SSL/TLS, most notably the Heartbleed [62] vulnerability (CVE-2014-0160). This allowed security researchers to extract secret keys, username and passwords from SSL/TLS traffic.

D. Deduplication

In [4], Harnik et al describe both file level and block level deduplication that is implemented by cloud storage providers. By eliminating redundant data, this enables cloud providers to maximise utilisation of the available hardware and storage drives. Harnik et al's paper shows that this opens up a side-channel, leaking information about both the existence of certain files, and the contents of files.

E. Legal and regulatory requirements

In [14] Kamara & Lauter outline three areas in which storing static data on cloud platforms can have legal and regulatory implications. If a company has to store certain personal information, such as medical or financial records, there are usually laws regarding the safe guarding of this data. Kamara et al states that therefore "the use of a public cloud storage service can involve significant legal risks" as "organizations are often held responsible for the actions of their contractors". Similarly, with different jurisdictions and geographic restrictions on storage of data, it can also be impossible to know in which region your data resides on cloud platforms. Lastly, Kamara et al explain how data on cloud platforms can be subject to subpoenas without notifying the user. As providers store the data of multiple users on the same disks, this has the potential to exposing the data of users who are not subject to investigation.

F. Data retention

A further point brought up by Kamara & Lauter [14] is that there are no guarantees that on requesting deletion of a file from cloud storage, that it is completely removed from all servers. Kamara et al describes this in the context of an issue relating to regulations on data retention and destruction, but it also an issue if an organisation simply does not want it's data to be retained indefinitely in a way that it cannot manage or guarantee destruction.

III. Client-Side Encryption as a key mitigation

Due to all the above concerns about privacy of data on cloud systems, any static data must be not therefore be stored in plaintext or transmitted across networks unencrypted. Cryptographic keys should also not be stored on the same system as the encrypted data. The key mitigation that solves nearly all these risks when storing static data, is strong client-side encryption, with keys securely stored on the local system or at least away from cloud infrastructure. With data stored on cloud platforms fully encrypted, any breach or compromise of the service is mitigated. Client-side encryption is not a silver bullet however – encryption can be broken, particularly through improper implementation or use of unsuitable algorithms. It will also therefore be necessary to understand the different cryptographic methods and the attacks against them.

There are three main requirements of client-side encryption that must be satisfied for such a mitigation to be successful – confidentiality, authentication and ensuring integrity of data.

A. Confidentiality

The encrypted file plaintext should be sufficiently scrambled such that it is impossible to differentiate from randomly generated binary data. Even if the attacker has multiple plaintext-ciphertext pairs (i.e. known-plaintext attacks), or is able to obtain ciphertexts for any plaintext (i.e. chosen-plaintext attacks), it should be computationally infeasible to brute force the key or break the algorithm. This requires a key of suitable length and an algorithm that contains sufficient complexity in the operations that it performs.

B. Authentication

It is vital that any encrypted data is also authenticated. This provides assurance that the encrypted data was encrypted with a particular key. Under the assumption that only the sender has access to the encryption key, it is possible to authenticate that the data came from the sender. Authentication also makes it possible to determine if a particular ciphertext has been modified in any way, so provides message integrity. In cloud storage scenarios, where the client does not have full control over the data, it is vital that such message tampering is detected.

Authentication can be provided by use of a Message Authentication Code (MAC) – in one of three possible configurations: Encrypt-and-MAC, MAC-then-Encrypt or Encrypt-then-MAC [38]. In [38] two types of authenticity/integrity are considered: 1. integrity of plaintexts – that it is computationally infeasible to produce an authenticated ciphertext from a plaintext that was not encrypted by the sender, and 2. integrity of ciphertexts that is infeasible to produce any kind of valid ciphertext, even if a previous plaintext is known. The conclusion of this paper is that only Encrypt-then-MAC provides both full privacy and both types of integrity if a strong, unforgeable MAC is used.

A useful feature of integrity of plaintexts as above is that it means chosen-ciphertext attacks are no longer possible. An authentication algorithm will recognise when a ciphertext is malformed, that is, it is has not been produced with a valid key or by the correct encryption algorithm and will refuse to decrypt it.

C. Integrity

Authentication and integrity in this context are generally used synonymously, however there exists attacks, namely forking or rollback attacks [16, 30] in which it's possible to have authentication without integrity. This is in a situation where there may be different versions of authenticated documents or information. The attacker restores or rolls back a previous, older, but authenticated version of a file. The older document will successfully decrypt and authenticate, however it is out of date. This introduces a new notion of integrity as a version or time-based integrity, different from the integrity of a single file, as provided by Authenticated Encryption algorithms, or Encryption algorithms with an additional MAC.

In order to also ensure version integrity, additional metadata about the latest version must be stored in some way.

IV. Cryptographic attacks

(incomplete) (place after Confidentiality, Auth, Integrity?) explain about different types before the attacks make sense

- Attacks on the algorithm – the computational effort required to brute force the algorithm.
- Side channel – such as timing attacks or EM/Power attacks the particular implementation, rather than the algorithm itself [41,44]
- decryption oracle/chosen-ciphertext attacks - Under confidentiality only schemes, since the binary data is indistinguishable from random bits, this opens up the possibility that decryption can be attempted on any carefully chosen binary data. By analyzing the results, information from this can be gained to break the cipher.

V. Existing solutions

(incomplete)

There exists a number of existing solutions for encrypting data on cloud storage.

Cloud providers themselves offer encryption options. For example AWS S3 allows server side encryption, using either AWS managed keys or customer provided keys. While Amazon states that customer provided keys are removed from memory once data has been encrypted [54], there is a fatal flaw in that the key has to be sent to AWS with each upload and download request. For at least the time of encryption and decryption, the key is available plaintext in the server memory. This opens the possibility for many external and internal attacks as above. As all encryption is handled server side, there is no guarantee that files are actually stored encrypted with the provided keys, and there remains the possibility for tampering unless we implement some client side authentication and integrity checks. Even worse, with AWS managed keys, the encryption keys are stored at rest on AWS servers.

There are a number of cryptographic methods that can provide both proof of storage and proof of retrievability without actually downloading the files [25, 34, 37]. However these typically are not possible with pure storage based cloud solutions, e.g. AWS S3, or Google Cloud Storage as they require additional code execution on the cloud server in a challenge-response format.

There is a solid range of Linux utilities to backup to cloud storage. Restic [link] provides a snapshot based backup system, and rclone is an rsync like clone tool to sync files to cloud providers. Both support encryption of files.

Rclone – uses older XSalsa20 rather than XChaCha20, no additional protection against rollback attacks, filename encryption is lacking – does not hide directory structure and identical filenames/directory name encrypt to identical strings

Boxcryptor etc, utilising existing solutions such as Dropbox etc.

VI. Requirements capture

The overall goal is to backup and/or sync files, utilising strong client-side encryption with secure defaults. There should be, in general, no possibility to store files in an insecure

way. The utility should be easy to use, providing a high-level interface that abstracts key handling and encryption functions in the background. Significant error checking and handling is also required to prevent data loss.

The program should have the ability to run as daemon, with the ability watch certain files or directories for changes on the local file system in the background. Any changes or new files should be automatically encrypted and uploaded to cloud storage. In the case where a directory with a recursive flag is added, the file system watcher should also add/watch all subdirectories and files recursively as well.

The design should be modular such that any cloud platform that supports an API allowing upload/downloading can be easily added.

- Utilise strong client-side authenticated encryption with secure defaults. There should be, in general, no possibility to store files in an insecure way.
- Prevent rollback/forking attacks.
- Hide completely filename and directory structure on remote storage.
- Providing an easy to use high-level interface that abstracts key handling, encryption functions and low level operations.
- Daemon service with the ability to watch certain files or directories on the local file system, and automatically upload any changes or new files to cloud storage. There should be a flag to recursively watch directories and it's subdirectories.
- Modular code such that any cloud platform that supports an API allowing upload/downloading can be easily added.
- Performant. Data upload and retrieval should not be significantly impaired.

VII. Program design and architecture

- 4 main functions, high level architecture
  - Watch directories/files
  - Encrypt
  - Upload to cloud
  - Restore
- High level class design
- Multi-threading ability
- The choice of C++
  - Most cryptographic libraries are written in C
  - Good interaction with the Linux filesystem
  - Excellent multi-threading support
  - Performant
- Index design

VIII. Cryptographic implementation and design

A common adage regarding encryption in software development is "don't roll your own" – in other words use algorithms that have undergone significant cryptanalysis, testing and development [51]. Encryption should not rely on obscurity, i.e. by hiding the implementation to be effective, it should be computationally unfeasible to break even if the full implementation is known. In a similar vein it is preferable to use implementations from well-known and tested libraries rather than implement the algorithms yourself. Even using well known and tested libraries, there is still the potential to incorrectly and unsafely use some of these in code [41]. For example, if nonces are reused with some algorithms then we "tend to lose confidentiality of messages [...] and authenticity tends to collaspe completely for all messages" [quote from 52, also see 35].

todo: Stream cipher vs Block cipher here and general description of modes here e.g. feedback mode

See [16] identical plaintext blocks encrypting to identical ciphertext can be used for file analysis

There are two main types of encryption to consider. Either asymmetric i.e. public key encryption or symmetric, where a single encryption key is used and shared between parties. When considering file encryption, symmetric key encryption is preferred as it is much less computationally intensive [17]. A hybrid method can be used by exchanging symmetric keys via public-key encryption, such as via a Diffie-Hellman exchange.

A. Deciding on suitable Authenticated-Encryption algorithms

As established above, authenticated encryption is essential with any cloud based application to prevent tampering of encrypted files [52]. This requires an algorithm that combines authentication with encryption, i.e. Authenticated-Encryption with Associated Data (AEAD) algorithms, or alternatively, use a standard, un-authenticated encryption mode with a MAC in the Encrypt-then-MAC format. Generally AEADs operate faster over large amounts of data, since a separate MAC takes extra time to compute, especially if the MAC is suitably secure.

There are also practical considerations when encrypting files, particularly for encrypting large files. Generally speaking to encrypt large files it will not be possible, or desirable to hold an entire copy of the file in memory, so it must be possible to stream the contents of the file to the encryption algorithm, or split the file into chunks to encrypt separately.

There is also an issue relating to scenarios where ordering is important, as described by Langley [52]. Splitting a large file into chunks will require establishing and checking the order or chunks, so messages cannot be recomposed in the wrong order or truncated without failing. As Langley states, there are no existing standards or practice for this so a "roll your own" implementation has the possibility for being insecure.

The most popular AEAD algorithms are AES in EAX, CCM and GCM modes, and Salsa20/ChaCha20-Poly1305. AES-GCM is the fastest of the AES variants, however it has the limitation that only roughly 350GB of data can be encrypted with a single key [53]. Further, the nonces used are short. While this mode is used extensively in TLS, it is not particularly suitable for file encryption. A custom chunking implementation would need to be written to resolve the issue of chunk re-ordering as above.

Salsa20-Poly1305 variants here – refer to Bernstein and other papers 39 to 50 & RFC7539

1) Choosing the correct encryption library: (incomplete) There are a large number of encryption libraries and algorithms to chose from. Three of the most established C and C++ libraries are the OpenSSL library, Crypto++ and NaCL (in particular libsodium).

OpenSSL is FIPS 140-2 validated – a U.S. government security standard. The Authenticated-Encryption with Associated Data (AEAD) schemes supported are AES in EAX, CCM and GCM modes. As these modes do not support splitting large files into chunks this will have to be handled separately.

Crypto++ supported AEADs are AES in GCM, CCM and EAX modes, ChaCha20Poly1305 and XChaCha20Poly1305.

NaCL, in particular the libsodium implementation provides the most suitable functions.

See [41] particularly - The security impact of a new cryptographic library.pdf https://libsodium.gitbook.io/doc/secret-key_cryptography/secretstream "The same sequence encrypted twice will produce different ciphertexts."

B. Filename encryption

For full confidentiality, encrypted files will need to be stored on cloud storage with hidden file names and paths. Both the directory structure, and filenames will leak information about the stored files so the ideal scenario is a completely flat file structure on the cloud with encrypted or otherwise obfuscated filenames that can be restored when downloading.

One option is to encrypt the path and filename with a secret key, then encode with base64 or base32 into filename a compatible string. A suitable algorithm must be chosen such that different nonces are used, otherwise identical filenames will encrypt to identical paths, leaking information about files that share the same filename or path. As a benefit this allows the possibility to restore files without the need for an index.

Alternatively we can hash the path/filename or file with a salt value. A unique salt value per file would be required otherwise identical filenames or files would hash to identical filenames. This would leak information as above, and additionally open the possibility for an attack via rainbow tables by pre-computing hashes to all possible filenames. A fixed, or short salt would allow an attacker to compute a rainbow table for every possible salt and filename, so in order for this to computationally infeasible, a long, unique hash per file is required. As the hash is a one way function, an index file would be required to track which hashes map to which file and also store the salt separately. The hash could however be used as a checksum and offer protection against rollback attacks (see integrity above), and assuming we have a list of the filenames with salt used we can recreate the names of encrypted files.

The most secure option would be to store files under completely random strings that have no connection to original filename or file. This removes the possibility of cryptanalysis of the filenames. In this option an index is required/vital and must be kept safe and securely backed up. Corruption or loss of the index would result in being unable to restore the original filenames/directory structure, however this is also the case with hashed filenames. As such this is the preferred option over hashes as there is no connection between the filename of the encrypted data and the original file.

IX. Local security mitigations

Todo. Assumes the host system is secure. Uses Unix filesystem permissions to restrict access to critical parts, e.g. key and index.db. However if local system is already compromised, then files also are already compromised.

X. Testing and benchmarks

Todo

XI. Critical analysis

Todo

XII. Limitations

- Multi-user access required
- This approach only works with static data, unable to compute on the data stored on the cloud this way

XIII. Future work/improvements

- Improvements to the index file system – use a Searchable Encryption such as outlined in [14]
- Stream encryption output stream directly to remote, without saving temporary files on the file system
- Mount remote storage on Linux with fusermount - handle wrapping encrypt/decrypt around this