Dark Cloud: A Secure File System

by Brando Miranda, Marcel Polanco and Pedro Cattori

December 13, 2013

Abstract

Dark Cloud is an encrypted file system that aims to store encrypted data remotely in an unstrusted remote server. The service seeks to provide confidentiality to the user's data and data integrity. The untrusted server will not be able to learn about the data being stored nor the names of the files. Furthermore, if the server attempts to tamper with any of the files, due to the cryptographic protocol and signatures being used in Dark Cloud, the user will be able to detect any changes immediately. At a high level, Dark Cloud will have a set of different keys that protect each file and using these keys is how sharing is impemented. The keys for each file are also stored at the server, which means that we need to secure this information before its sent to the server. This is done by generating using user keys derived from the user's password, username and the path to the file we want to secure.

1 Introduction

The goal of our design is to minimize the amount of information and keys the user has to remember in order access all his data from the server while still providing data integrety and confidentiality. Every file stored in the untrusted server will be stored in a format we will refer in this paper to as a *secure file*[1]. The secure file will be computed as follows before being sent to the server:

$$E_{K_{AES-CBC}}[\text{plaintext},\ Sign_{RSA}[Hash[\text{plaintext}]]] \tag{1}$$

For every secure file there will be two set of keys; one set generated for AES-CBC and another generated from RSA. It is important to note that for every file, there will be one *validation key* and one *signing key*. These will be asymmetric because this allows us to enforce a MAC policy.

Dark Cloud aims to store as much data as possible in a secure manner in the untrusted sever. When a user creates a file, keys are generated randomly for that file and are used to make a secure version of the file as in equation (1). These randomly generated keys are created only on the client and the untrusted server never sees the plaintext version of the keys. The keys securing the user's

[1]Unix treats directories and files in a very similar manner. Therefore, we will treat them in similar way when making a secure directory. The only difference is as follows: within a seperate, secure file stored with the directory are the contents that the directory would output with an ls command. This will make detecting an unauthorized change in a directory easy because we will simply check if the directory content in the server matches that of the accompanying secure file that we encrypt and sign.

file are made into a *secure keychain file* that is stored on the untrusted server. The *secure keychain file* is encrypted and signed using the user's password and salt as in equation (1). The server will not be able to get the original file the user created because he does not know the password that *locked* the keychain file, thus he cannot get the file's keys to *unlock* the user's secure file. Furthermore, the server will not even be able to tell the difference between a keychain file or a normal file because everything it gets will be encrypted (including the names of the files/directories).

Now one should question, why is having a keychain file even neccesary? Why is everything not just encrypted with the *user key* dervived from his password? The reason is as follows: if a user wanted to share a file then he would either have to share his password or the user key, giving complete access to the owner's file system. To avoid this, we choose to have randomly generated keys for each file such that they can be individually shared with other users and only provide access to one file/directory at a time. Enforcing read/write permissions is as simple as limiting the keys sent within the *keychain file*; we only share the AES-CBC key and the verification key from RSA to decrypt and verify the contents, inherently enforcing read permissions. Intrinsically, to allow write permissions, we share all keys for that specific file. The shared user can encrypt the keychain file with his own user key for the owner to place back in the containing directory.

We will also enforce that directory contents do not change without the user's permission. We do this by making a Dark Cloud version of the directory contents file (i.e. the data that would be outputed with an ls command) known as a *directory content file*. With this, our file system tracks a directory's contents from the server and verifies by matching them to those in the *directory content file*. Since the protocol could be known by the server, each *directory content file* will be made into a secure file as in equation (1) and stored in the server (for it to be possible to guarantee data integrity and confidentility).

All files and directory names will remain confidential and thus will be encrypted. They will be encrypted with the AES-CBC key corresponding to the file that it corresponds to. To change names, the user has to re-encrypt the names. TODO: explain why names don't have to be signed

Due to the way that we are making secure files as in equation (1), the untrusted server will not be able change the user's file without Dark Cloud detecting the changes because the cryptographic signature will not verify. Furthermore, it will not be able to understand the contents of the files because they will be encrypted. One last nice property that CBC provides is of forward diffussion that makes changing bits in the encrypted version hard to manipulate in order to get changes as desired (that can be made even harder to do with applying AES-CBC on both directions or simply using BitLocker's diffusion techniques). An obvious benefit of this method is that malicious users trying to change files are also easily detected.

2 Definitions and terms

To make the rest of the paper easier to read, I will define the following terms (some of which have been already used on the previous section). The details of how keys are generated can be found on the DarkCloudCryptoLib library.

keychain: a key chain is the set of keys used to make some data secure. It comprises of the AES-CBC key, the IV key for encryption and the RSA verification and signing key. The main purpose of keychains is to make secure files and data as in equation (1).

lock: locking a file is a function any keychain can perform that makes a secure file with the set of keys the keychain has by using equation (1).

unlock: unlocking a file is a function any keychain can perform that inverts equation (1) and returns the plaintext. In the process of this, the signature is verified.

user keychain: a user keychain is a keychain generated from the user's password, username and pathname to the file to lock. The goal of it is to secure key files. When the user keychain is created it generates its key_{AES} key, its iv vector and its RSA keys by using a salt from the sha256 on the username, the PBKDF2 algorithm on the password and the path of the file it intends to lock. The details of how is generated can be found on the DarkCloudCryptoLib library.

file keychain: a set of keys generated randomly used to lock and protect the actual user contents. It is used to lock the user's actual data content on his files. This keychain has the same keys as the user keychain, but the important difference is that the keys are generated randomly. Since the user does not have to remember these, they are stored as a secure key file in the untrusteded server. The details of how they are generated can be found on the DarkCloudCryptoLib library.

directory keychain: a directory keychain is used to make a secure file of the directory content file

secure file: a data file protected and locked by equation (1).

directory content file: the directory content file is the Dark Cloud version of the directory content that the unix system would output with a ls command. It's use is to detect that the server's directory content matches with the what the user has created and deleted.

secure directory content file: the locked version of a directory content file.

keychain file: has the same content as a file keychain, but stored as a string in a file that will be stored at the server after being locked.

directory keychain file: are the keys that lock the directory content file. i.e. the keys that lock and secure the ls file for a directory

Dark Cloud Client: is the file system software installed at the user's side that enfornces the Dark Cloud protocol for storing files at the server. It is also in charge of creating files and securing them with the cryptography in equation (1).

3 Protocol Design

3.1 Overview

The main idea behinde the protocol is that every file the user makes has a corresponding keychains for protecting that file. In turn, the keys for each file are stored at the untrusted server and protected from it by the keys made from user's passwords, username and file pathname. If a user wants to share a file he shares the appropriate keys with his friend and his friend in turn securely stores those files with the original keys. However, his friend will now store the his with his own password and username so that he can lock and unlock the the shared file.

3.2 Creating files

When a file is created by the user the Dark Cloud Client will create a keychain file that will be used to lock the new file. After locking, it is safe to send the secure contents of the user's data. Furthermore, the keychain file will be locked with the user's password, username and path to file. This will protect the keychain file from any untrusted entity. Only the user that can create a user keychain can unlock this keyfile. Furthermore, the path will aid to detect if the server responses with the contents of a different keychain file.

It will also be neccesary to update the directory contents at the server. Therefore, the directory contents that Dark Cloud is tracking will be compared to the one the server have and make sure the server is creating thing that we command. This directory content file will also have its own directory keychain that is protected with the user's keychain. Notice that this means that there will be two directory contents in Dark Cloud, one being tracked by the server and another version that the server cannot change tracked by the Dark Cloud Client.

3.3 Deleting files

When deleting files the procedure will be very similar to creating one. A delete command will be send to the server and an update to the directory content file will be required. This way we track that the untrusted server is deleting the files that we command.

3.4 Sharing files

When sharing file with another user, the user of the original file will give him the plaintext of the keychain file for the file he wants to share. Depending what privileges it wants to share it will share different keys in the key chain:

reading files: if the owner of the file only wants to give read access, then the other user must be able to decrypt equation (1) and verify it but not sign any new plaintexts. Therefore a keychain file version only containing the AES-CBC keys and the RSA validation key will be shared with the other user. The new user will need to store this shared keys in some way. Therefore, it will lock the shared keys in a secure file with its own user keychain. This way it can always re-read the file that it wants.

writing files: if the owner of the file wants to give write access to the file, then the other user must be able to decrypt equation (1), verify it and any sign new plaintexts. Therefore, the user will share the plaintext of the keychain file with all the keys. Again the new user will store his own version of the keychain file on the server protected by his own user keychain. Notice that the actual shared file is always locked with the same keychain.

Notice that because the user gives keys (and therefore privilege) to other user's when it wants set permission for files, it implements a Mandatory Access Control (MAC) permission scheme to control access to its files.

3.5 Directories

Directories are supported by Dark Cloud. The way we support them is to treat them as similar as possible as unix does. The import idea to understand is that the untrusted server will keep track of two directory contents. One will be the usual one that unix would be tracking. The second one will be tracked by Dark Cloud and it will be encrypted. At all time the files that the server is following should match the ones Dark Cloud is following. Otherwise, the untrusted server has tampered with the directory. The will be two verification steps to verify that the directory was not tampered with. First, we will cryptographically verify that the one that Dark Cloud is following has not been changed. Secondly, we will compare the directory contents the server is following with the ones it should have, and if it matches, we conclude that the directory has not been tampered. Obviously, if the server decides to lie to us about what it has and it says it has a file when it really doesn't, this will be easily detected if the user tries to access that file again.

3.6 Naming for Files and Directories

3.6.1 Confidentiality of names

File names remain confidential and they are encrypted with the file keychain for that file or directory keychain for that directory. Filenames for keychain files are encrypted with the user's keychains.

3.6.2 Renaming

When a file is renamed the directory content that Dark Cloud is tracking has to change. Therefore, the Dark Cloud Client will change the name in this file. It will also issue a rename unix command to the server and will change the name at the server. Again, if the directory content file that Dark Cloud is tracking does not match the one the server is tracking, we can easily detect that the server is not changing file names as we wanted. To change the file name, the Dark Cloud Client simply performs:

$$new\_encrypted\_name = E_{K_{AES-CBC}}[new\_name]$$

and issue the unix change name command to the server.

4 Security

The main idea about the security of Dark Cloud is that every file is locked with some keychain. When sharing a file, a user will share it's file keychain and the receiving user will make sure that that keychain remains secure from the server by locking it with its own user key. If the user has the correct keychain corresponding to the file it is trying to access, then the cryptographic signature should verify (if it was not tampered with). Otherwise, the keychain does not match and it cannot possibly verify.

4.1 Cryptography

Recall equation (1)

$$keychain.lock(plaintext) = E_{K_{AES-CBC}}[\text{plaintext},\ Sign_{RSA}[Hash[\text{plaintext}]]] \tag{1}$$

Unlocking is the following operation:

$$keychain.unlock(secure\_file) = plaintext \tag{1}$$

Included in the unlocking operation is a verification step (not shown). If that verification step does not pass then even if you can decrypt the file you are not guaranteed its correct. Therefore, unlock will return an error.

4.2 Detecting unauthorized behavior by the untrusted server

Detecting changes to files and directories is simple. Since the untrusted server does not have access to the plaintext version of the keychains, it cannot decrypt the protected data nor it can create signatures. Therefore, if the untrusted server tries to flip bits of the cipher text (due to the properties of AES-CBC) it will change the plaintext too and changed signature and plaintext are extremely unlikely to verify (since he doesn't even have the RSA keys he does not know how to change the signature).

4.3 Detecting unauthorized behavior by the untrusted user

The case where there is an untrusted user is nearly identical to an untrusted server. If the untrusted user does not have read privileges, then we can treat this case the same as an untrusted server. However, if we do give read privileges to the user, then he can see the signature and verify it. If the server is trusted but the user is not, then unless he has the signing key dervided user's password and the encryption key, then the user is unable to do anything malicious that we cannot detect.

4.4 Detecting when the Server supplies the wrong file

Dark Cloud is able to identify the case when the untrused server gives the incorrect secure file even though we asked for a different one. The reason is because the user at his end will request the keychain file too. Unless the untrusted server is able to supply the correct version of the keychain file, then we will get keys that do not match for the file that we got and therefore verification will not pass. However, notice that this argument does not apply for when we request for a secure file keychain. If the user gives a wrong secure file keychain, then how would be able to detect this with the password and username alone? You can't unless use the unencrypted path to the keychain file that you requested. So that is what we do. We use the unencrypted path to the keychain file, so that every keychain file also has dependency on their unencrypted name. This way, if the server gives us the wrong keychain file, Dark Cloud would create the wrong user keychain (because it depends on the unencrypted name of the file) and then create wrong keys, which would not pass the verification step.
The way to think about this is, its similar to bitlocker, where they use sector numbers, which can be thought as the name of the object being encrypted, but we instead use the path and the unencrypted name which are unique to each file. Notice that the path is neccesary because in the unfortunate accident that we have two files with the same name, they might be locked with the same user keychain. We got around this by using the path.

5 Implementation

5.1 Dark Cloud Crypto Library

The Dark Cloud Crypto Library is a custom made library using other cryptographic libraries to implement locking files and unlocking files.

5.2 Generating key chains

Every keychain is made from two set of keys: one from RSA and the other from AES-CBC keys.

5.2.1 User keychain:

A user keychain is called DCTableKey (due to backward compatibility in our own code). This object holds all the keys for implementing equation (1). The Key_{AES} is generated from a salt and the user's password. The salt is generated from the sha256 of the username and the PBDKDF2 is used on the password to generate this key. The iv vector is generated from Key_{AES} and the sha256 of path to file. In turn these two are used by the PBDKDF2 library to make the iv vector. This make the user keychain dependent by the user's password and the name of the object that is planned to be encrypted.

5.2.2 file and directory keychain:

The algorithm for making the keys for this is the same as the user keychain. The difference is the input that this algorithm takes. When generating the keys istead of passwords and usernames we instead use os.urandom to make keys for each file.

5.3 Dark Cloud Client

The Dark Cloud Client controls a shell, our Crypto library, and an HTTP server together in order to accomplish the security goals of Dark Cloud all on the client-side.

DCClient implements the Dark Cloud Client as described above.

DCWorkingDirectory keeps the state of the working directory on the client-side as the fileserver may have multiple users at different working directores.

DCDir modularizes the updates made to the directory content file when there is a write made to a directory (i.e. creating/deleting a file/directory at that directory).

DCClientParser implements the shell that takes input from the user.

DCCrytoClient implements an object representation of our customized crypto-library.

5.4 Future work and bugs

Dark Cloud has the capability to share files but this is not currently an option available to the user. With the implemented security scheme, it is possible to share files by communicating the unlocked *file keychain* to a desired user and the encrypted path to the file to be shared over a secure channel. The user should then respond by encrypting the keyfile with their own *user keychain* and sending it back to the original shared file owner to write to the file's parent directory. Using the originally sent encrypted file path, the shared user can access the shared file by using their key written in the same parent directory by the file owner.

Encoding schemes that "play well" with our HTTP client/server, our encryption library, and Unix filenames should be employed as currently the urllib.quote() encoding we use occasionaly results in unparsable input for our encryption library. Currently, there is a bug in our HTTP client/server implementation resulting in Nonetype headers being received at the server despite the input provided by the client. Writing more durable adapters for a variety of fileservers would make our client more appealing. Due to the design of Dark Cloud, none of the security is done server-side, thus making adapters possible for virtually any fileserver.