Amazon S3 RC4 Encryption - Final Report

**Introduction:**

S3FS-fuse is a powerful tool for Unix that allows users to mount an Amazon S3 bucket via Fuse. When files are added to the local bucket, they are automatically uploaded and available on the web client. Files uploaded on the web client are likewise automatically downloaded to the local bucket as long as it is still mounted. For my Operating System’s final project, I implemented my own encryption program that can encrypt and decrypt files using the RC4 algorithm and integrated it within S3FS. This results in every file being automatically encrypted before they are uploaded from the bucket and automatically decrypted when they are downloaded to the bucket. In addition, my encryption is compatible with OpenSSL and is salted which adds much more security and protection against attacks such as using Rainbow Tables. Although S3FS is written from a mixture of C and C++, majority of the code I wrote was in C and relied on other C libraries.

**Project Goals:**

The goals for this project were to first, create a standalone implementation of a RC4 encryption program, and second, to integrate it into an S3FS installation. For the standalone implementation, the program should prompt the user to enter several parameters. The parameters are the operation (encryption or decryption), the password to encrypt/decrypt with, the input file, and the output file. For the S3FS integration, files that are moved into the mounted bucket should automatically be encrypted before they are uploaded to S3 and decrypted when they are downloaded from S3. Both the standalone and integration should be compatible with OpenSSL and should salt the encryption. If the program is compatible with OpenSSL, it means that a file encrypted with my implementation should be able to be decrypted through OpenSSL’s rc4 decryption method and vice versa. Salting encryptions works by adding a randomly generated set of bytes to the beginning of the file to ensure a unique hash. This adds additional security and helps protect the encrypted data from being susceptible to attacks involving Rainbow Tables. These tables contain precomputed hashes of the most common passwords (password, 12345, etc). By adding the salt to the beginning of the file before it is encrypted, the files are guaranteed to generate a unique hash rendering Rainbow Tables useless.

**System Information:**

Host:

For this project, I used a Lenovo IdeaPad 530s-14ARR running Windows 10 (64 bit) build 1809. This model contained 8GB of RAM, a 256GB SSD and was powered by an AMD Ryzen 5 CPU and Vega 8 GPU.

Guest:

Development and testing occurred in a virtual Linux environment powered by Oracle VirtualBox 6.0.4. The specifications of the virtual machine included 2GB RAM allocation, a 20GB Virtual Disk Image (VDI), and running Ubuntu 18.0.4.2 (64-bit).

**Tools and Packages:**

The tools and packages used for this project were S3FS-Fuse, Amazon S3, OpenSSL, and g++. S3FS-Fuse was installed directly from its github repository (<https://github.com/s3fs-fuse/s3fs-fuse>) and had the latest versions of S3FS (1.85) and Fuse (3.5.0) at the time. OpenSSL was already pre-installed in my Ubuntu build and was running 1.1.0g. I already had g++ installed as I needed it for previous labs to compile .cpp files. The version I was using for this project was 7.3.0. Finally, this project uses Amazon’s S3 cloud storage as a backend and therefore, you must register an account at <https://aws.amazon.com/s3/>.

**Design:**

The architecture of the modified S3FS comprises of four key components. The first three components are within the RC4 function that I implemented in the beginning of the fdcache.cpp file. The first component is reading the file that is passed into the function. The contents of the file are extracted to a buffer and examined to check whether the file is encrypted or decrypted. If the file is encrypted, it needs to be decrypted so the salt hash is read from the file. If the file is decrypted, a salt hash is generated. The second component is EVP which hashes the password and applies the salt hash. This is what enables the password to be compatible with OpenSSL. The third component is passing the buffer with the file contents and the newly hashed password into the RC4 functions to encrypt/decrypt the file content. The result is written back to the original file. The last component of the modified S3FS architecture is calling my RC4 function in the Load() and RowFlush() functions. These are two instances where the file is passed before it is uploaded to Amazon S3 and after a file is downloaded to the local bucket.

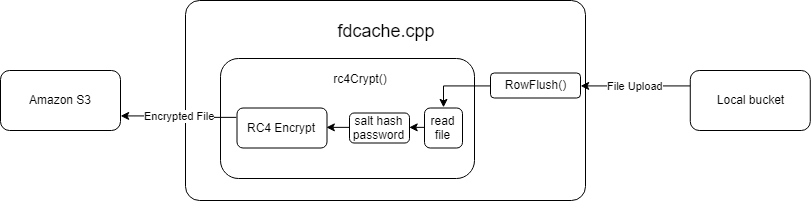


Figure 1: Example of bucket uploading file to S3 and encrypting it

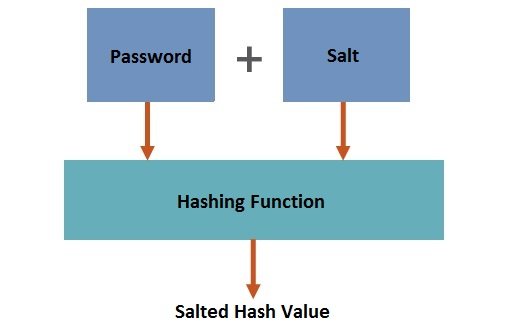


Figure 2: Second component showing password combined with salt passed to hash function to get salted hashed password

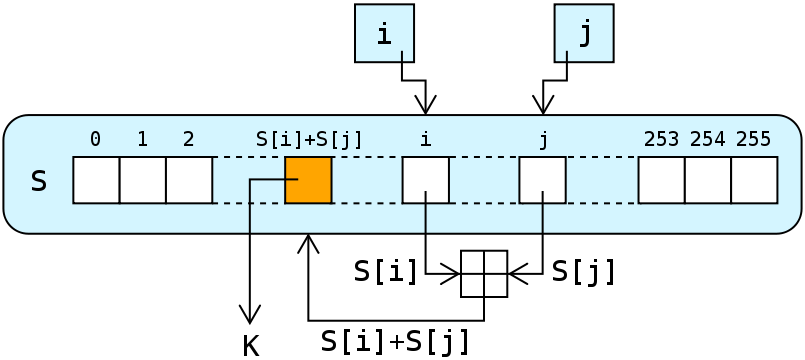


Figure 3: How the RC4 algorithm encrypts data in the third component

**Integration and Implementation:**

RC4 Standalone:

In my standalone program, four input parameters are expected to be given by the user. The first is the operation (encryption/decryption). The second is the desired password. The third is the input file and the fourth is the output file. The first part of the program is to open the input file with read only permission and open the output file with truncate, create, and read/write permissions. The contents of the input file are then extracted into a large buffer array. If the encryption option was selected, the program will generate 8 random bytes and store them into an array called “salt”. An array called “magic” which contains the phrase “Salted\_\_” and the salt array are written in the first 16 bytes of the output file. If decryption is selected, the file cursor will be set to the 8th byte and reads the next 8 bytes. This contains the salt hash and is stored in the salt array. A function named EVP\_BytesToKey() is then called with the encryption type, the hash type, the salt array, the password, the password length, and the new password given as parameters. In this case, we are using the RC4 encryption type and sha256 hash type (better than MD5) and are given a hashed version of our password in newpassword. This function handles both the salting and the hashing of the password which enables OpenSSL compatibility. If the decryption option was selected, the file cursor will be moved to the 16th position and update the length of the file from that point to the end. This is because we want to ignore the “Salted\_\_” and salt hash part that are in an encrypted file once we decrypt it. The rc4\_setkey() function is then called which sets the RC4 key using the hashed password saved in newpassword. The RC4() function is then called with the RC4 key, the buffer containing the input file content, and the output buffer given as parameters. The encrypted/decrypted result is stored in the output buffer. The contents of the output buffer are then written to the output file.

S3FS Integration:

In the fdcache.cpp, I implemented my RC4 encryption function. The first part of my function is reading the file passed in. I do this by extracting the contents of the file into a large buffer array. RC4 is a symmetric cipher meaning encryption and decryption normally work the same way. I implemented salting however, and therefore must determine whether the file has already been encrypted or not. I do this by reading the first 8 bytes of the file and determine whether the “Salted\_\_” message is present. If it exists, it means the file is already encrypted and must be decrypted. The salt hash contained in the next 8 bytes are read and stored in another array and an encryption flag is set to false. If the message is not present, the file needs to be encrypted. A random set of 8 bytes are generated and stored as the new salt hash. The next part is EVP which hashes the password with sha256 and applies the salt. EVP will hash the original password with the given parameters into a new password that will be used in the actual RC4 encryption. This hashing with sha256 is what also makes the implementation compatible with OpenSSL. The last part is the RC4 encryption. Using precompiled libraries in crypto, two rc4 functions are called. The first one is rc4\_setkey() which sets the key using the newly hashed password. The second one is rc4() which performs the actual encryption taking the rc4 key, the file size, the buffer, and the output buffer as parameters. The encrypted contents of the file are stored in the output buffer which is then written to the original file. If the operation was an encryption, then “Salted\_\_” and the salt hash are added to the beginning of the file. I call my function in the Load() and RowFlush() functions further down in the fdcache.cpp so that the file is encrypted/decrypted before it is uploaded and after it is downloaded.

**Source Code:**

RC4 Standalone:

#include <openssl/rand.h>

using namespace std;

int main(int argc, char \*argv[]){

//User should type executable + encryption/decryption + password + inputFile + outputFile

//Need to add more error checking. Does file input exist? Too many arguments given?

if(argc<5) //Check for sufficient arguments

{

printf("Error, too few arguments. Give encryption/decryption (e/d) + password + inputFile + outputFile");

return 0;

}

//Define Variables

int fd = open(argv[3],O\_RDONLY); //Open input file

int fd2 = open(argv[4], O\_CREAT | O\_TRUNC | O\_WRONLY, 0644); //Open output file with appropriate permissions

const int buffSize = 4000000;

unsigned char buffer[buffSize]; //Buffer that will hold file (4mb limit currently)

int fileSize = read(fd,buffer,buffSize); // Size of file

unsigned char outputBuffer[buffSize]; // Buffer that will hold encrypted/decrypted file

unsigned char salt[8]; //Char array storing salt hash

unsigned char \*salty; //Char array pointer that will either point to salt generated (encryption) or salt hash read from file (decryption)

char \*pass = argv[2]; //char pointer to password

static const char magic[] = "Salted\_\_"; //Char array containing whether file has been salted or not

bool encrypting = false; //Flag if encryption operation

//Determine encryption or decryption

//If encryption selected

if(strcmp(argv[1],"e") == 0)

{

RAND\_bytes(salt, sizeof(salt)); //Generate random bytes in salt array

write(fd2,magic,8); //Write the term Salted\_\_ in first 8 bytes of output file

write(fd2,salt,8); //Write the salted hash in next 8 bytes of output file

encrypting = true; //Set flag true

}

//If decryption selected

else if (strcmp(argv[1],"d") == 0)

{

lseek(fd,8,SEEK\_SET); //Move cursor to 8th position (where salt hash starts)

read(fd,salt,8); //Read 8 bytes from the 8th postion (contains the salt hash) and store in salt array

}

//If invalid option selected

else

{

printf("Error: You must enter either e for encryption or d for decryption %s",argv[1]);

return 0;

}

salty = salt; //Set salty pointer to salt array that contains the salt hash (salt hash generated if encryption selected or salt hash read from file if decryption selected)

unsigned char newPassword[16];//[EVP\_MAX\_KEY\_LENGTH];

EVP\_BytesToKey(EVP\_rc4(),EVP\_sha256(), salty, (const unsigned char \*)pass, strlen(pass), 1, newPassword, NULL); //Create Openssl compatable password with salt stored in "newpPassword"

int newPassLength = 16; //Size of newPassword array

//Encrypt using RC4

if(!encrypting) //If decrypting, skip the salt part

{

lseek(fd,16,SEEK\_SET); //Set cursor to after salted stuff

fileSize = read(fd,buffer,buffSize); //Update fileSize

}

RC4\_KEY key;

RC4\_set\_key(&key, newPassLength, (const unsigned char\*)newPassword); //Set RC4 key using newpassword

RC4(&key, fileSize, (const unsigned char\*)buffer, outputBuffer);

lseek(fd2,0,SEEK\_END); //Move cursor to end of output file

write(fd2,outputBuffer,fileSize); //Move contents there

return 0;

}

S3FS Integration:

Fdcache.cpp:

My RC4 Function:

#include <openssl/rc4.h>

#include <openssl/evp.h>

#include <fstream>

#define passKey "hotdog"

using namespace std;

//Arhum's RC4 Implementaion

void rc4Crypt(int fd){

//Define Variables

const int buffSize = 4000000;

unsigned char buffer[buffSize]; //Buffer that will hold file (4mb limit currently)

char \*pass = passKey;

unsigned char enc[8];

int fileSize = read(fd,buffer,buffSize); // Size of file

unsigned char outputBuffer[buffSize]; // Buffer that will hold encrypted/decrypted file

unsigned char salt[8]; //Char array storing salt hash

unsigned char \*salty; //Char array pointer that will either point to salt generated (encryption) or salt hash read from file (decryption)

static const char magic[] = "Salted\_\_"; //Char array containing whether file has been salted or not

bool encrypting = false; //Flag if encryption operation

//Determine encryption or decryption based on whether salt exists

lseek(fd,0,SEEK\_SET); //Start at beginning of file

read(fd,enc,8); //Read the first 8 bytes

if(memcmp(enc,magic,8)==0)//If the first 8 bytes of the file say "Salted\_\_", file needs to be decrypted

{

lseek(fd,8,SEEK\_SET); //Move cursor to 8th position (where salt hash starts)

read(fd,salt,8); //Read 8 bytes from the 8th postion (contains the salt hash) and store in salt array

}

//File needs to be encrypted

else

{

RAND\_bytes(salt, sizeof(salt)); //Generate random bytes in salt array

encrypting = true; //Set encrypting flag to true

}

salty = salt; //Set salty pointer to salt array that contains the salt hash (salt hash generated if encryption selected or salt hash read from file if decryption selected)

unsigned char newPassword[16];

EVP\_BytesToKey(EVP\_rc4(),EVP\_sha256(), salty, (const unsigned char \*)pass, strlen(pass), 1, newPassword, NULL); //Create Openssl compatable password with salt stored in "newPassword"

int newPassLength = 16; //Size of newPassword array

//Encrypt using RC4

if(!encrypting) //If decrypting, skip the salt part

{

lseek(fd,16,SEEK\_SET); //Set cursor to after salted stuff

fileSize = read(fd,buffer,buffSize); //Update fileSize

}

RC4\_KEY key;

RC4\_set\_key(&key, newPassLength, (const unsigned char\*)newPassword); //Set RC4 key using newpassword

RC4(&key, fileSize, (const unsigned char\*)buffer, outputBuffer); //Store enrypted/decrypted version of file in outputBuffer

if(encrypting) //If encrypting

{

fileSize = read(fd,buffer,buffSize) - 1; //Update fileSize

ftruncate(fd,0);//clear file

lseek(fd,0,SEEK\_SET); //Move cursor back to beginning of file

write(fd,magic,8); //Write the term Salted\_\_ in first 8 bytes of output file

write(fd,salt,8); //Write the salted hash in next 8 bytes of output file

write(fd,outputBuffer,fileSize); //Move encrypted file contents stored in outputBuffer back into file

}

else //If decrypting

{

lseek(fd,0,SEEK\_SET); //Move cursor to start of output file (want to overwrite the salted part)

write(fd,outputBuffer,fileSize); //Move decrypted file contents stored in outputBuffer back into file

ftruncate(fd, fileSize); //Remove extra anything after decrypted file contents

}

}

FdEntity Load Modification:

int FdEntity::Load(off\_t start, size\_t size)

{

.

.

.

// initialize for the area of over original size

if(0 < over\_size){

if(0 != (result = FdEntity::FillFile(fd, 0, over\_size, (\*iter)->offset + need\_load\_size))){

S3FS\_PRN\_ERR("failed to fill rest bytes for fd(%d). errno(%d)", fd, result);

break;

}

// set modify flag

is\_modify = false;

//Arhum implementation

rc4Crypt(fd);

}

// Set loaded flag

pagelist.SetPageLoadedStatus((\*iter)->offset, static\_cast<off\_t>((\*iter)->bytes), true);

}

PageList::FreeList(unloaded\_list);

}

return result;

}

FdEntity RowFlush Modification:

int FdEntity::RowFlush(const char\* tpath, bool force\_sync)

{

int result = 0;

S3FS\_PRN\_INFO3("[tpath=%s][path=%s][fd=%d]", SAFESTRPTR(tpath), path.c\_str(), fd);

if(-1 == fd){

return -EBADF;

}

AutoLock auto\_lock(&fdent\_lock);

if(!force\_sync && !is\_modify){

// nothing to update.

return 0;

}

//Arhum implementation

rc4Crypt(fd);

// If there is no loading all of the area, loading all area.

size\_t restsize = pagelist.GetTotalUnloadedPageSize();

if(0 < restsize){

if(0 == upload\_id.length()){

// check disk space

if(ReserveDiskSpace(restsize)){

// enough disk space

// Load all uninitialized area

result = Load();

FdManager::FreeReservedDiskSpace(restsize);

if(0 != result){

S3FS\_PRN\_ERR("failed to upload all area(errno=%d)", result);

return static\_cast<ssize\_t>(result);

}

}else{

// no enough disk space

// upload all by multipart uploading

if(0 != (result = NoCacheLoadAndPost())){

S3FS\_PRN\_ERR("failed to upload all area by multipart uploading(errno=%d)", result);

return static\_cast<ssize\_t>(result);

}

}

}else{

// already start multipart uploading

}

}

.

.

.

Fdcache.h

class FdEntity

{

.

.

.

private:

static int FillFile(int fd, unsigned char byte, size\_t size, off\_t start);

void Clear(void);

int OpenMirrorFile(void);

bool SetAllStatus(bool is\_loaded); // [NOTE] not locking

//bool SetAllStatusLoaded(void) { return SetAllStatus(true); }

bool SetAllStatusUnloaded(void) { return SetAllStatus(false); }

public:

explicit FdEntity(const char\* tpath = NULL, const char\* cpath = NULL);

~FdEntity();

void Close(void);

//Arhum Implementation

void rc4(int fd);

bool IsOpen(void) const { return (-1 != fd); }

bool IsMultiOpened(void) const { return refcnt > 1; }

int Open(headers\_t\* pmeta = NULL, ssize\_t size = -1, time\_t time = -1, bool no\_fd\_lock\_wait = false);

bool OpenAndLoadAll(headers\_t\* pmeta = NULL, size\_t\* size = NULL, bool force\_load = false);

int Dup();

const char\* GetPath(void) const { return path.c\_str(); }

void SetPath(const std::string &newpath) { path = newpath; }

int GetFd(void) const { return fd; }

.

.

.

**Future Improvements:**

Both my standalone program and my S3FS integration met all the functional requirements for this project. In addition, I was able to get both my standalone and S3FS integration encryption compatible with OpenSSL and applying a salt hash to every encryption. For future improvements, there are three main areas that I would focus on.

File Size Limitation:

The first area would be addressing the limitation in file size my encryption supports. Currently, my implementation only supports files smaller than 4MB. This is because I extract the file contents to a temporary buffer of fixed size 4000000 (~4mb). There are several approaches I could explore to get around this. The most obvious would be to create a loop that would continuously extract content from the file to a buffer. If the buffer is full and data remains in the file, the buffer contents could be offloaded to a vector and the next 4000000 bytes of the file would be read into the buffer. This would continue until the file is completely extracted.

Redundancies:

Although my program works, there are definite instances of redundancies that reduce the efficiency of the program. In the interest of debugging and eliminating any possible ambiguities, I often wrote commands and operations that were likely unnecessary. For example, I call the function lseek() several times to get the file cursor at the expected location. After calling functions such as read(), the cursor automatically moves to the end of the specified count parameter so in many cases, the cursor was already in location I wanted it at.

Error Checking:

A good program is capable of handling errors and giving the user helpful feedback on what they did wrong. While my standalone does give a message if the user enters too few arguments, it doesn’t check if there are too many arguments or if the arguments are in the correct order. The standalone also assumes that the given input file exists in the directory and therefore will always try to read it even if it doesn’t exist. In addition, if the file is larger than 4mb, the program will not warn the user and therefore the encryption will cut off the data.

**Summary:**

This project was very challenging in terms of the actual coding required, but also in regards to the various configuration steps and processes required for developing in the Linux environment. Previously, I had only ever developed on Windows 10 and always used some fully fledged IDE such as Visual Studios or PyCharm. For this project, I needed to learn how the different components of S3FS-Fuse interacted with each other by running it in debug mode and examining traces. This was a much more laborious and ambiguous way of understanding the code than what I am normally used to. I also spent quite a bit of time learning how to use the Linux terminal and the necessary configurations for different operations. One example was when I developed my standalone program, I was getting compiler errors stating that the rc4\_setkey() and RC4() functions I had referenced did not exist. It was a very frustrating process, but I eventually learned that I needed to manually link the precompiled crypto library when I was compiling my program using g++. I also had many issues trying to get my bucket mounted and unmounted. Although I have had a lot of experience developing with C++, most of the coding for this project relied on C functions and logic I had never heard of. This project helped me get a much better understanding of how to use Linux and develop programs for it. Much of the C functions used were low-level commands that operated on the byte level. In addition, this project taught me a lot about cyber security in terms of different encryption algorithms, different hashing algorithms, OpenSSL, and salt hashing.